Lab 1 - Pointers Recap and Abstraction

1)

Aim:

1. Write a C++ program to find the sum of 'n' integers using only pointers. Maintain proper boundary conditions and follow coding best practices.

Algorithm:

Input: n – number of integers ,**n** integers

Output: sum – sum of n integers

1)Get the value of n

2)Initiate while loop if n!=0

->Get new integer and add to sum

->Decrement n

3)Print sum

<u>**Time complexity analysis-**</u> **O(n)**

Program:

//Write a C++ program to find the sum of 'n' integers using only pointers. Maintain proper boundary conditions and follow coding best practices.

```
#include <stdio.h>

#include <stdlib.h>


int main() {

   int *n,*sum,*temp;

   n = (int *)malloc(sizeof(int));

   sum = (int *) malloc(sizeof(int));
```

```
    *sum = 0;

    printf("Enter the value of n: ");
    scanf("%d",n);

    if(*n<=0) {
        printf("\nn should be positive\n");
    }
    else {
        printf("Enter the integers: \n");
        while(*n !=0) {
            scanf("%d",temp);
            *sum += *temp;
            *n -= 1;
        }
        printf("The sum is: %d",*sum);
    }

    return 0;
}
```

Screenshot of Output:

```
Enter the value of n: 6
Enter the integers:
10
20
30
40
50
60
The sum is: 210
```

2.

Aim:

To implement a calculator that performs various arithmetic operations.

Algorithm:

1. Calculator Program

INPUT: opt => Option of the user, **num1** => First Integer, **num2** => Second Integer.

OUTPUT: Resultant Value.

1. Start Menu options.
2. **Case Add:**
   Print Sum of **num1** and **num2**.
3. **Case SUBTRACT:**
   Print Difference of **num1** and **num2**.
4. **Case MULTIPLY:**
   Print Product of **num1** and **num2**.
5. **Case DIVIDE:**
   Print Quotient of **num1** and **num2**.
6. **Case EXIT:**
   Exit Menu

Time Complexity:

a. Addition – O(1)
b. Subtraction – O(1)
c. Multiplication – O(1)
d. Division – O(1)

Program:

Header:

```
#include <stdio.h>


int add_no(int *n1 , int *n2){

  return *n1 + *n2;
```

```
}

int sub_no(int *n1 , int *n2){
    return *n1 - *n2;
}

int mul_no(int *n1 , int *n2){
    return *n1 * *n2;
}

int div_no(int *n1 , int *n2){
    return (*n1 / *n2);
}
```

CPP file:

```
#include "Q2.h"

int main() {
    int n1= 1,n2 =1 ,choosen_opt =1;

    while(choosen_opt != 6) {
        printf("\nCALCULATOR \n 1)SET \n 2)ADD \n 3)SUBTRACT \n 4)MULTIPLY \n 5)DIVIDE \n 6)EXIT
\n");
        printf("Choose a number: ");
        scanf("%d",&choosen_opt);

        switch(choosen_opt) {
            case 1:
                printf("Enter the value of 1st number: ");
                scanf("%d",&n1);

                printf("Enter the value of 2nd number: ");
```

```
            scanf("%d",&n2);

          case 2:
            printf("Sum of %d and %d is: %d",n1,n2,add_no(&n1,&n2));
            break;

          case 3:
            printf("Difference Of %d and %d is %d",n1,n2,sub_no(&n1,&n2));
            break;

          case 4:
            printf("Multiplication Of %d and %d is %d",n1,n2,mul_no(&n1,&n2));
            break;

          case 5:
            printf("Division Of %d and %d is %d",n1,n2,div_no(&n1,&n2));
            break;

          case 6:
            printf("APPLICATION CLOSED\n");
            break;

          default:
            printf("Not available");
            break;
          }
    }
}
```

Output:

```
CALCULATOR
 1)SET
 2)ADD
 3)SUBTRACT
 4)MULTIPLY
 5)DIVIDE
 6)EXIT
Choose a number: 1
Enter the value of 1st number: 10
Enter the value of 2nd number: 20
```

```
Choose a number: 2
Sum of 10 and 20 is: 30
CALCULATOR
 1)SET
 2)ADD
 3)SUBTRACT
 4)MULTIPLY
 5)DIVIDE
 6)EXIT
Choose a number: 3
Difference Of 10 and 20 is -10
```

```
Choose a number: 4
Multiplication Of 10 and 20 is 200
CALCULATOR
 1)SET
 2)ADD
 3)SUBTRACT
 4)MULTIPLY
 5)DIVIDE
 6)EXIT
Choose a number: 5
Division Of 10 and 20 is 0
CALCULATOR
 1)SET
 2)ADD
 3)SUBTRACT
 4)MULTIPLY
 5)DIVIDE
 6)EXIT
Choose a number: 6
APPLICATION CLOSED
```

Lab 2 - Searching and Sorting

1)

Aim:

To implement a program to search for the presence of a number in an array.

Algorithm:

1. Linear Search in array

   INPUT: N => Number of elements, **search** => Element to search for, **array** => array of elements.

   OUTPUT: Index of Element if found , **-1** if element not found

   1. Get number of elements **N**
   2. Create the array of elements with for loop insertion.
   3. Get the element to search for **search**.
   4. Initialize a for loop to access array elements.
      1. If current element = **Search** , print index
      2. Else continue
   5. Print element not found in array

Time complexity -> O(n)

Code:

```
// Write a C++ program to search for the presence of a number in an array.

#include <stdio.h>

#include <stdlib.h>


int main() {

   int *a,*size,*element,*i;

   size = (int *)malloc(sizeof(int));

   element = (int *)malloc(sizeof(int));

   i = (int *)malloc(sizeof(int));

   printf("Enter number of elements ");

   scanf("%d",size);

   a = (int *)malloc(sizeof(int) * (*size));


   for ( *i = 0 ; *i < *size ; (*i)++) {

      printf("Enter %d element",*i + 1);
```

```
        scanf("%d",(a + (*i)));
    }
    printf("Enter a element to search ");
    scanf("%d",element);
    for(*i = 0 ; *i < *size ; (*i )++ ) {
        if(*(a + (*i)) == *element) {
            printf("ELEMENT FOUND in %d position\n",(*i) + 1);
            return 0;
        }

    }
    printf("Element not found\n");


}
```

Output:

```
Enter number of elements 6
Enter 1 element1
Enter 2 element2
Enter 3 element3
Enter 4 element4
Enter 5 element5
Enter 6 element6
Enter a element to search 4
ELEMENT FOUND in 4 position
```

2)

Aim:

To implement a program to search for the presence of a number in an array.

Algorithm:

1. Ascending order bubble sort
   INPUT: array => array of integers, **N** => Array size
   OUTPUT: array => Sorted array in ascending order.
   1. Initialize **first** for loop from **0** to **N-1**
      1. Initialize **second** for loop from **first** to **N**.
         1. If **first > second**:
            1. Swap **First** and **Second**.
   2. Return **array**

2. Descending order bubble sort
   INPUT: array => array of integers, **N** => Array size
   OUTPUT: array => Sorted array in ascending order.
   1. Initialize **first** for loop from **0** to **N-1**
      1. Initialize **second** for loop from **first** to **N**.
         1. If **first < second**:
            1. Swap **First** and **Second**.
   2. Return **array**

Time Complexity:
The program was implemented using bubble sort algorithm and it has a time complexity of O(n^2) in the worst case, where n is the number of elements in the array.

1. Ascending – O(n^2)
2. Descending – O(n^2)

Code:
//2. Write a C++ menu-driven program to sort an array of numbers in ascending or descending order. After you write the function for sorting, search online and find what type of sorting you have done in your code.

```
#include <stdio.h>

#include <stdlib.h>
```

```
void ascending(int *arr,int *n){
    int *i,*first,*second,*tmp;
    first = (int *)malloc(sizeof(int));
    second = (int *)malloc(sizeof(int));
    tmp = (int *)malloc(sizeof(int));
    for(*first = 0;*first <*n -1;(*first)++){
        for(*second = *first + 1;*second < *n;(*second)++){
            if(*(arr + (*first)) > *(arr + (*second))){
                *tmp = *(arr + (*first)) ;
                *(arr + (*first)) = *(arr + (*second));
                *(arr + (*second)) = *tmp;
            }
        }
    }
    for(*i = 0; *i < *n;(*i)++){
        printf("%d ",*(arr + (*i)));
    }
    printf("\n");
}
void descending(int *arr,int *n){
    int *i,*first,*second,*tmp;
    first = (int *)malloc(sizeof(int));
    second = (int *)malloc(sizeof(int));
    tmp = (int *)malloc(sizeof(int));
    for(*first = 0;*first <*n -1;(*first)++){
        for(*second = *first + 1;*second < *n;(*second)++){
            if(*(arr + (*first)) < *(arr + (*second))){
                *tmp = *(arr + (*first)) ;
                *(arr + (*first)) = *(arr + (*second));
                *(arr + (*second)) = *tmp;
```

```
            }
        }
    }
    for(*i = 0; *i < *n;(*i)++){
        printf("%d ",*(arr + (*i)));
    }
}

int main() {
    int option = 0;
    int *a,*n,*i;
    n = (int *)malloc(sizeof(int));
    i = (int *)malloc(sizeof(int));


    while(option != 4) {
    printf("MENU \n 1)INPUTARRAY \n 2)ASCENDING \n 3)DESCENDING \n 4)EXIT \n ");
    printf("Enter the option ");
    scanf("%d",&option);
    switch(option) {
        case 1:
            printf("Enter the size of the array ");
            scanf("%d",n);
            a = (int *)malloc(sizeof(int) * (*n));
            for( *i = 0;*i< *n;(*i)++) {
                printf("Enter a[%d]",*i);
                scanf("%d",a + (*i));

            }
            break;
```

```
        case 2:
          ascending(a,n);
          break;

        case 3:
          descending(a,n);
          break;

        case 4:
          printf("PROGRAM ENDED");
          break;

        default:
          break;

    }

    }
```

Output:

```
MENU
 1)INPUTARRAY
 2)ASCENDING
 3)DESCENDING
 4)EXIT
 Enter the option 1
Enter the size of the array 6
Enter a[0]1
Enter a[1]2
Enter a[2]3
Enter a[3]4
Enter a[4]5
Enter a[5]6
```

```
 Enter the option 2
1 2 3 4 5 6
MENU
 1)INPUTARRAY
 2)ASCENDING
 3)DESCENDING
 4)EXIT
 Enter the option 3
6 5 4 3 2 1 MENU
 1)INPUTARRAY
 2)ASCENDING
 3)DESCENDING
 4)EXIT
 Enter the option 4
PROGRAM ENDED
```

3)

Aim:

Program to search for the given roll number in a sorted array.

Algorithm:

INPUT: **N** - Number of students, **find** - The roll number to search for.
OUTPUT: arr - Sorted array of roll numbers in ascending order**,**Position of the searched roll number (if found)

1. Get the number of students **N** from the user.
2. Create an array **arr** of size **N** to store the roll numbers.
3. Initialize a loop from **0** to **N-1**:
    1. Get the roll number from the user and store it in array index.
4. Sort in Ascending Order:
    1. Initialize **first** to 0.
    2. Initialize a loop from **first** to **N-2**:
        1. Initialize **second** to **first + 1**.
        2. Initialize a loop from **second** to **N-1**:
            1. If **arr[first] > arr[second]**, swap **arr[first] and arr[second]**
5. Print the sorted array of roll numbers.
6. Get the roll number to search for from the user as **find**.
7. Search For Roll Number:
    1. Initialize **Start** to **0** and **End** to **N-1**.
    2. While **Start <= End**:
        1. Calculate mid as (Start + End) / 2.
        2. If arr[mid] == find, print the position (mid + 1) and return mid.
        3. If arr[mid] < find, update Start to mid + 1.
        4. If arr[mid] > find, update End to mid - 1.
    3. If the loop exits without finding the roll number, print "Roll number not found" and return mid.

Time complexity:

Binary search : O(logn)

Code :

```
// Program to find a number

#include <stdio.h>

#include <stdlib.h>


int main(){

   int *arr,*n,*search,*i;

   n = (int *)malloc(sizeof(int));
```

```
    search = (int *)malloc(sizeof(int));
    i = (int *)malloc(sizeof(int));
    printf("Enter the size of the array ");
    scanf("%d",n);
    arr = (int *)malloc(sizeof(int) * (*n));
    for(*i  = 0 ; *i<*n ; (*i)++){
        printf("Enter the number in arr[%d] ",(*i) + 1);
        scanf("%d",(arr + (*i)));
    }
    printf("Enter the number to search ");
    scanf("%d",search);
    for(*i = 0 ; *i < *n ; (*i)++){
        if(*(arr + (*i)) == *search){
            printf("Element is at %d position\n",*i + 1);
            return 0;
        }
    }
    printf("Element not found \n");
    return 0;
}
```

Output:

```
Enter the size of the array 6
Enter the number in arr[1] 1
Enter the number in arr[2] 2
Enter the number in arr[3] 3
Enter the number in arr[4] 4
Enter the number in arr[5] 5
Enter the number in arr[6] 6
Enter the number to search 4
Element is at 4 position
```

WEEK 3: List ADT: Array implementation Date:07/02/24

Aim:

To write a program to implement List ADT.

Functions used:

1. insertbeg();
2. display();
3. append();
4. insertpos();
5. deletebeg();
6. pop();
7. deletepos();
8. search();

Algorithm:

1) Insert at beginning

```
Input: num,arr
Output: 1 if element is added else 0
    If cur=size-1
        Return 0
    Else
        Temp=cur
        While temp!=0
            Arr[temp+1]=arr[temp]
            Temp--
        Arr[0]=num
        Inc cur
        Return 1
```

2)Insert at end

```
Input:num,arr
Output:1 if number is added else 0
    If cur=size-1
        Return 0
    Else
        Arr[cur]=num
        Return 1
```

3)Insert at end

```
Input:num,pos,arr
Output:1 if element is added and 0 if not
```

```
If cur=size-1  or pos>cur
      Return 0

Else
   Temp=pos
   Repeat until temp>=cur
      Arr[temp+1]=arr[temp]
   Arr[pos]=num
   Cur++
    Return 1
```

4)Delete at beginning

Input : arr
Output: 0 or 1

```
If cur=-1
   Return 0
Else
   Temp=0
   Repeat until temp=cur
       Arr[temp]=arr[temp+1]
   Cur - -
   Return 1
```

5) Delete at end

Input: arr
Output:0 or 1

```
If cur=-1
   Return 0
Else
   Cur - -
    Return 1
```

6) Delete at position

Input: arr
Output:0 or 1

```
If cur=-1
    Return 0;
Else:
    If pos=0
       Call delbeg
    Else if pos=cur
        Call delend
    Else
         Temp=pos
```

```
Repeat until temp<=cur
      Arr[temp]=arr[temp+1]
Cur- -
Return 1
```

7) Search

Input: num,arr
Output: 0 or 1

```
I=0
Repeat until I is not greater than or equal to cur
      If arr[i]=num
       Return 1

 Return 0
```

8)Display

Input: arr
Output: displaying the elements

```
If cur =-1
   Display array is empty
else
  Temp=0
  Repeat until temp <=cur
     Display arr[temp]
```

Time complexity:

1) Insertion at beginning : O(n)
2) Insertion at end : O(1)
3) Insertion at position : O(n)
4) Deletion at beginning: O(n)
5) Deletion at end: O(n)
6) Deletion at position: O(n)
7) Search: O(n)
8) Display: O(n)

Code:

```
/*
1. Write a C++ menu driven program to implement List ADT using arrays. Maintain proper boundary conditions and follow good coding practices.
*/
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
class List
{
    int arr[SIZE];
    int cur;
    public:
        List()
        {
            cur = -1;
        }
        int insertbeg(int);
        void display();
        int append(int);
        int insertpos(int,int);
        int deletebeg();
        int pop();
        int deletepos(int);
        int search(int);
};

int main()
{
    List l1;
```

```
int choice, num;
int app;
while (1)
{
    printf("\nEnter \n1. Insert Begin\n2. Append\n3. Insert Position");
    printf("\n4. Delete Begin\n5. Pop\n6. Delete Position");
    printf("\n7. Search\n8. Display\n9. Exit");
    printf("\n Enter a choice:");
    scanf("%d",&choice);
    switch (choice)
    {
    case 1:
        printf("\n Enter the number to insert ");
        scanf("%d",&num);
        if(l1.insertbeg(num))
        {
            printf("\n %d successfully inserted.",num);
        }
        else
        {
            printf("\n Failed to insert %d. The list is full",num);
        }
        break;
    case 2:
        printf("\n Enter the number to append");
        scanf("%d",&app);
        if(l1.append(app)) {
            printf("\n %d SUCCESSFULLY APPENDED ",app);
        }
        else {
            printf("Failed to append");
```

```
    }
    break;

case 3:
    int num;
    int pos;
    printf("\n Enter the number to insert and its position ");
    scanf("%d %d",&num,&pos);
    if(l1.insertpos(num,pos)) {
        printf("Sucessfully inserted %d",num);
    }
    else{
        printf("Empty in middle or position greater than size");
    }
    break;
case 4:
    if(l1.deletebeg()) {
        printf("Successfully deleted");
    }
    else {
        printf("Empty list");
    }
    break;

case 5:
    if(l1.pop()) {
        printf("Successfully poped");
    }
    else {
        printf("Empty list");
    }
```

```
    break;

case 6:
    int position;
    printf("Enter the position of the list to delete ");
    scanf("%d",&position);
    if(l1.deletepos(position)) {
        printf("Sucessfully deleted");
    }
    else if(l1.deletepos(position) == 2) {
        printf("The position is empty");
    }
    else {
        printf("List empty");
    }
    break;

case 7:
    int searchelement;
    printf("Enter the element to search in the list ");
    scanf("%d",&searchelement);
    if(!l1.search(searchelement)) {
        printf("Not Found");
    }
    break;
case 8:
    l1.display();
    break;

case 9:
    exit(0);
```

```
            break;
        default:
            printf("\n Enter a valid choice\n");
            break;
        }

    }
    return 0;
}

//Method to insert a number in begining of the list
int List::insertbeg(int num)
{
    if(cur==SIZE-1)
    {
        return 0;
    }
    else if(cur==-1)
    {
        cur = 0;
        arr[0]=num;
        return 1;
    }
    else
    {
        for(int i=cur;i>=0;i--)
        {
            arr[i+1]=arr[i];
        }
        cur = cur + 1;
        arr[0]=num;
```

```
        return 1;


    }
}
//Method to append a number to last of the list
int List::append(int num) {
    if(cur == SIZE-1) {
        return 0;
    }
    else if (cur == -1) {
        cur = 0;
        arr[0] = num;
        return 1;
    }
    else {
        cur = cur + 1;
        arr[cur] = num;
        return 1;


    }
}
//Method to insert a value in its position and shifting the previous values to the rightbrave
int List::insertpos(int num , int pos) {
    if(pos > cur + 1 || pos > SIZE) {
        return 0;
    }
    else if(cur == -1 && pos == 0){
        cur = 0;
        arr[0] = num;
        return 1;
    }
```

```
    else {
        for(int i=cur;i>=pos;i--){ // If cur < pos for loop will not be triggered and the value will automatically be appended
            arr[i+1]=arr[i];
        }
        cur = cur + 1;
        arr[pos]=num;
        return 1;

    }
}

//Method to delete the beginning value
int List::deletebeg() {
    if(cur == -1) {
        return 0; // LIST EMPTY
    }
    else {
        for(int i = 0;i < cur + 1 ; i++) {
            arr[i] = arr[i+1];

        }
        cur = cur - 1;
        return 1;

    }
}

//Method to pop the last element
int List::pop() {
```

```
    if (cur == -1) {
        return 0;
    }
    else {
        //arr[cur] = 0; //0 ACTS AS SPECIAL NUMBER AND USER CANNOT ADD 0 TO THE LIST HEREAFTER
        cur = cur - 1;

    }
}

//Method to delete element in the position of the list
int List::deletepos(int pos) {
    if (cur == -1 || pos > cur) {
        return 0;
    }
    else {
        for( int i = pos + 1 ; i < cur + 1; i++ ) {
            arr[i] = arr[i+1];
        }
        cur = cur - 1;
        return 1;
    }
}

//Method to search for the element in the list
int List::search(int searchelement) {
    for(int i = 0 ; i< (cur + 1) ; i++) {
        if(arr[i] == searchelement) {
            printf("The element %d is found in the %d position",searchelement,i);
            return 1;
        }
```

```
    }
    return 0;
}

//Method to display the contents of the list
void List::display() {
{
    printf("\nThe contents of the list are:");
    for(int i=0;i<=cur;i++)
    {
        printf("%d ",arr[i]);
    }
}
}
```

Output:

```
Enter
1. Insert Begin
2. Append
3. Insert Position
4. Delete Begin
5. Pop
6. Delete Position
7. Search
8. Display
9. Exit
 Enter a choice:1

 Enter the number to insert 1

 1 successfully inserted.
```

```
Enter a choice:1

Enter the number to insert 2

2 successfully inserted.
```

```
Enter a choice:8

The contents of the list are:2 1
```

```
Enter the number to append3

3 SUCCESSFULLY APPENDED
```

```
Enter a choice:8

The contents of the list are:2 1 3
```

```
Enter a choice:3

Enter the number to insert and its position 4
3
Sucessfully inserted 4
```

```
Enter a choice:8

The contents of the list are:2 1 3 4
```

```
Enter a choice:4
Successfully deleted
```

```
Enter a choice:8

The contents of the list are:1 3 4
```

```
Enter a choice:5
Successfully poped
```

```
Enter a choice:8

The contents of the list are:1 3
```

```
Enter the position of the list to delete 0
Sucessfully deleted
```

```
Enter a choice:7
Enter the element to search in the list 1
The element 1 is found in the 0 position
```

```
9. Exit
 Enter a choice:9
PS C:\Users\kesav\
```

<u>WEEK 4:LIST ADT:SINGLY LINKED LIST</u> Date:14/02/24

Aim:

To Write a C++ menu-driven program to implement List ADT using a singly linked list.

Functions used and time complexity:

1. int insert_beg(int num) : O(1)
2. int insert_end(int num) : O(n)
3. int insert_pos(int num , int pos) : O(n)
4. int del_beg() : O(1)
5. int del_end() : 0(n)
6. int del_pos(int pos) : O(n)
7. int search(int num) : O(n)
8. void display() : O(n)
9. void displayreverse() : O(n)
10. int reverselist() : O(n)
11. int size() : O(n)

Algorithms:

1.INSERT AT BEGINNING:

Input:num,head
Output: 0 or 1

```
If  head=NULL
   Head=newnode
   Return 1
Else
   Newnode->next=head
   Head=newnode
   Return 1
```

2.INSERT AT END:

Input:num,head
Output:0 or 1

```
If  head=NULL
   Head=newnode
   Return 1
Else
```

```
Temp=head
Repeat until temp->next!=NULL
    Temp=temp->next
Temp->next=newnode
Return 1
```

3.INSERT AT POS:

Input:num,pos,head
Output:0 or 1

```
If pos=0
  Call insertbeg()
Else
  Temp=head
  Repeat until temp->next!=NULL and pos!=1
     Temp=temp->next
     Pos- -
  Newnode->next=temp->next
   Temp->next=newnode
   Return 1
```

4.DELETE AT BEGINNING:

Input: head
Output: 0 or 1

```
If head==NULL
   Return 0
Else
   Head=head->next
   Return 1
```

5.DELETE AT END:

Input: head
Output:0 or 1

```
If head=NULL
   Return 0
Else
   Temp=head
   Repeat until temp->next!=NULL
       Temp2=temp
       Temp=temp->next
   Temp2->next=NULL
   Free(temp)
   Return 1
```

6.DELETE AT POS:

Input: pos,head
Output: 0 or 1

```
If head=NULL
   Return 0
Else
   Temp=head
   Repeat until temp->next!-NULL and pos!=1
      Temp2=temp
      Temp=temp->next
      Pos- -
   Temp2->next=temp->next
   Free(temp)
   Return 1
```

7.SEARCH:

Input: num,head
Output: 0 or 1

```
If head=NULL
   Return 0
Else
   Temp=head
   Repeat until temp->next!=NULL
      If temp->data=num
         Return 1
      Temp=temp->next
   Return 0
```

8.DISPLAY:

Input:head
Output: All the elements in the list

```
If head=NULL
   Display the list is empty
Else
   Temp=head
   Repeat until temp!=NULL
      Display temp->data
      Temp=temp->next
```

9.DISPLAY REVERSE:

Input:head

Ouput: elements displayed in reverse order

```
If head=NULL
  Return
Else
  Call disrev(head->next)
  Display head->data
```

10.REVERSELINK :

Input: head

Output: List is reversed (0 or 1)

```
If head=NULL
  Return 0
Else
  Initialize temp1 and temp2 to NULL
  Repeat until head!=NULL
     Temp2=head->next
     Head->next=temp1
     Temp1=head;
     Head=temp2
  Head=temp1
  Return 1
```

1.INSERT AT BEGINNING:

Input:num,head
Output: 0 or 1

```
If head=NULL
  Head=newnode
  Return 1
Else
  Newnode->next=head
  Head=newnode
  Return 1
```

2.INSERT AT END:

Input:num,head
Output:0 or 1

```
If head=NULL
  Head=newnode
  Return 1
Else
  Temp=head
  Repeat until temp->next!=NULL
     Temp=temp->next
  Temp->next=newnode
```

Return 1

3.INSERT AT POS:

Input:num,pos,head
Output:0 or 1

```
If pos=0
  Call insertbeg()
Else
  Temp=head
  Repeat until temp->next!=NULL and pos!=1
     Temp=temp->next
     Pos- -
  Newnode->next=temp->next
   Temp->next=newnode
   Return 1
```

4.DELETE AT BEGINNING:

Input: head
Output: 0 or 1

```
If head==NULL
   Return 0
Else
   Head=head->next
   Return 1
```

5.DELETE AT END:

Input: head
Output:0 or 1

```
If head=NULL
   Return 0
Else
   Temp=head
   Repeat until temp->next!=NULL
      Temp2=temp
      Temp=temp->next
   Temp2->next=NULL
   Free(temp)
   Return 1
```

6.DELETE AT POS:

Input: pos,head
Output: 0 or 1

```
If head=NULL
   Return 0
Else
   Temp=head
   Repeat until temp->next!-NULL and pos!=1
      Temp2=temp
      Temp=temp->next
      Pos- -
   Temp2->next=temp->next
   Free(temp)
   Return 1
```

7.SEARCH:

Input: num,head
Output: 0 or 1

```
If head=NULL
   Return 0
Else
   Temp=head
   Repeat until temp->next!=NULL
      If temp->data=num
         Return 1
      Temp=temp->next
   Return 0
```

8.DISPLAY:

Input:head
Output: All the elements in the list

```
If head=NULL
   Display the list is empty
Else
   Temp=head
   Repeat until temp!=NULL
      Display temp->data
      Temp=temp->next
```

9.DISPLAY REVERSE:

Input:head
Ouput: elements displayed in reverse order

```
If head=NULL
  Return
```

```
Else
  Call disrev(head->next)
  Display head->data
```

10.REVERSELINK :

Input: head

Output: List is reversed (0 or 1)

```
If head=NULL
  Return 0
Else
  Initialize temp1 and temp2 to NULL
  Repeat until head!=NULL
     Temp2=head->next
     Head->next=temp1
     Temp1=head;
     Head=temp2
  Head=temp1
  Return 1
```

Code:

```
/*

A. Write a C++ menu-driven program to implement List ADT using a singly linked list. Maintain proper boundary conditions and follow good coding practices. The List ADT has the following operations,


1. Insert Beginning

2. Insert End

3. Insert Position

4. Delete Beginning

5. Delete End

6. Delete Position

7. Search

8. Display

9. Display Reverse

10. Reverse Link

11. Exit

*/


#include <stdio.h>
```

```
#include <stdlib.h>

class List {
   struct Node {
      int data;
      struct Node *next;
   };

   struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
   struct Node *head;

   public:
      List() {
         head = NULL;
      }
      int insert_beg(int num);
      int insert_end(int num);
      int insert_pos(int num , int pos);
      int del_beg();
      int del_end();
      int del_pos(int pos);
      int search(int num);
      void display();
      void displayreverse();
      int reverselist();
      int size();



};
```

```
int main() {
   List l1;
   int choice;
   int num;
   int pos;
   while(1) {

      printf("\n SINGELY LINKED LIST \n");
      printf("\n 1. Insert Beginning \n 2. Insert End \n 3. Insert Position");
      printf("\n 4. Delete Beginning \n 5. Delete End \n 6. Delete Position");
      printf("\n 7. Search \n 8. Display \n 9. Display Reverse \n 10. Reverse Link \n 11. Exit");
      printf("\n Enter the choice ");
      scanf("%d",&choice);

      switch(choice) {
         case 1:
            printf("Enter the number ");
            scanf("%d",&num);
            if(l1.insert_beg(num)) {
               printf("\n Inserted successfully.");
            }
            else {
               printf("\n Insertion unsuccessful.");
            }
            break;

         case 2:
            printf("Enter the number ");
            scanf("%d",&num);
            if(l1.insert_end(num) == 2) {
               printf("Insertion successful.");
```

```
    }
    break;

case 3:
    printf("Enter the number and position to insert ");
    scanf("%d %d",&num,&pos);
    if(l1.insert_pos(num , pos)) {
        printf("Inserted successfully.");
    }
    else {
        printf("Position out of bounds");
    }

    break;

case 4:
    if(l1.del_beg()) {
        printf("Deleted successfully.");
    }
    else {
        printf("List is empty.");
    }
    break;

case 5:
    if(l1.del_end()) {
        printf("Deleted successfully.");
    }
    else {
        printf("The list is empty.");
    }
```

```
   break;

case 6:
   printf("Enter the position ");
   scanf("%d",&pos);
   if(l1.del_pos(pos)) {
      printf("Deleted successfully.");
   }
   else {
      printf("The list is empty.");
   }
   break;

case 7:
   printf("Enter the number ");
   scanf("%d",&num);
   printf("%d \n",l1.search(num));
   break;

case 8:
   l1.display();
   break;

case 9:
   l1.displayreverse();
   break;

case(10):
   if(l1.reverselist()) {
      printf("Successfully reversed.");
   }
```

```
            else {
                printf("The list is empty.");
            }
            break;


        }
    }

}

//Getting the size of the singly linked list.
int List::size() {
    struct Node *temp = head;
    int count = 0;
    if(head == NULL) {
        return 0;
    }

    else {
        while(temp != NULL) {
            count = count + 1;
            temp = temp -> next;
        }
        return count;
    }
}

//Inserting at the beginning of the singly linked list.
int List::insert_beg(int num) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
```

```
    newnode -> data = num;
    newnode -> next = head;
    head = newnode;
    return 1;

}

//Inserting at the end of the singely linked list.
int List::insert_end(int num) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    struct Node *temp = head;
    newnode -> data = num;
    newnode -> next = NULL;
    if(head == NULL) {
        head = newnode;
        return 2;
    }

    else {
        while(temp -> next != NULL) {
            temp = temp -> next;

        }
        temp -> next = newnode;
        return 2;
    }
}

//Inserting at a position in the singely linked list.
int List::insert_pos(int num , int pos) {
    int count = 0;
```

```
    struct Node *temp = head;
    struct Node *temp2;
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    newnode -> data = num;

    if(pos > (size())) {
        return 0;
    }
    else {
        if(head == NULL && pos == 0) {
            insert_beg(num);
        }
        else if(pos == 1) {
            temp = head -> next;
            head -> next = newnode;
            newnode -> next = temp;
            return 1;
        }
        else {
            while(count < pos - 1) {
                temp = temp -> next;
                count++;
        }
        temp2 = temp -> next;
        temp -> next = newnode;
        newnode -> next = temp2;
        return 1;
        }
    }
}
```

```
//Deleting the beginning of the singely linked list.
int List::del_beg() {
   struct Node *temp;
   struct Node *temp2;
   if(head == NULL) {
      printf("The list is empty.");
      return 0;
   }

   else {
      temp2 = head; //temp2 stores the memory address of 0 element
      temp = head -> next; //temp stores the memory of 1 element
      head = temp;
      free(temp2);
      return 1;
   }
}

//Deleting the end of the singely linked list.
int List::del_end() {
   struct Node *temp = head;
   if(head == NULL) {
      return 0;
   }
   else {
      while(temp -> next -> next != NULL) {
         temp = temp -> next;
      }
      temp -> next = NULL;
      free(temp -> next); //Frees the node in the memory address.
      return 1;
```

```
    }
}

//Deletes the element in that position in the singly linked list.
int List::del_pos(int pos) {
    struct Node *temp = head;
    struct Node *prev = head;
    int count = 0;
    if(head == NULL) {
        return 0;
    }

    else {
        while(count < pos && temp != NULL) {
            prev = temp;
            temp = temp -> next;
            count++;
        }
        prev -> next = temp -> next; //Links previous node to one after the other.
        return 1;
    }
}

//Searching the number in the singly linked list.
int List::search(int num) {
    struct Node *temp = head;
    int count = 0;

    while(temp -> data != num) {
        temp = temp -> next;
        count++;
```

```
    }
    return count;

}

//Displaying the data of each nodes in the singely linked list.
void List::display() {
    struct Node *temp = head;
    if(head == NULL) {
        printf("List empty");
    }
    else {
        while(temp != NULL) {
            printf("%d ",temp->data);
            temp=temp->next;
        }
    }
}

void List::displayreverse() {
    int arr[100];
    int count = 0;
    struct Node *temp = head;
    if(head == NULL) {
        printf("List empty");
    }

    else {
        while(temp != NULL) {
            arr[count] = temp -> data;
            temp = temp -> next;
```

```
            count = count + 1;
        }
    }

    for(int i = count-1 ; i >= 0 ; i--) {
        printf("%d\n",arr[i]);
    }
}

//Method to reverse List in the singely linked list.
int List::reverselist()
{
    if(head==NULL)
    {
        return 0;
    }
    struct Node *left = head;
    struct Node *temp1;
    struct Node *temp2;
    temp2=left->next;
    while(temp2!=NULL)
    {
        temp1=left;
        left=temp2;
        temp2=left->next;
        left->next=temp1;
    }
    head->next=NULL;
    head=left;

    return 1;
```

}

Output:

OUTPUT:

```
 SINGELY LINKED LIST

 1. Insert Beginning
 2. Insert End
 3. Insert Position
 4. Delete Beginning
 5. Delete End
 6. Delete Position
 7. Search
 8. Display
 9. Display Reverse
 10. Reverse Link
 11. Exit
 Enter the choice
1
Enter the number 1

 Inserted successfully.
```

```
 Enter the choice 1
Enter the number 2

 Inserted successfully.
```

```
 Enter the choice 1
Enter the number 3

 Inserted successfully.
```

```
 Enter the choice 1
Enter the number 4

 Inserted successfully.
```

```
 Enter the choice 8
4 3 2 1
```

Insertend

```
 Enter the choice 2
Enter the number 5
Inserted
```

```
 Enter the choice 8
4 3 2 1 5
```

insertpos

```
Enter the number and position to insert 6 5
Inserted successfully.
```

```
 Enter the choice 8
4 3 2 1 5 6
```

Deletebeg

```
 Enter the choice 4
Deleted successfully.
```

```
 Enter the choice 8
3 2 1 5 6
```

Delete end

```
 Enter the choice 5
Deleted successfully.
```

```
 Enter the choice 8
3 2 1 5
```

Deletepos

```
Enter the position 1
Deleted successfully.
```

```
Enter the choice 8
3 1 5
```

Search

```
Enter the choice 7
Enter the number 1
1 in the 1 index
```

```
Enter the choice 8
3 1 5
```

```
Enter the choice 9
5
1
3
```

```
Enter the choice 10
Successfully reversed.
```

```
Enter the choice 8
5 1 3
```

```
Enter the choice 11
PS C:\Users\kesav\OneDrive - SSN Trust\YEAR1_DStruct\CODE\LAB\LAB4(14-02-2024)(SINGLEYLINKEDLIST)>
```

2)

Aim:

To Write a C++ menu-driven program to implement List ADT using a singly linked list and use gethead() private member

Algorithm:

1)Insert Ascending

2)Merge

3)Display

Time complexity:

Insert ascending : O(n)

Merge : O(n+m)

Display :O(n)

Code:

Header file:

/*

B. Write a C++ menu-driven program to implement List ADT using a singly linked list. You have a gethead() private member function that returns the address of the head value of a list. Maintain proper boundary conditions and follow good coding practices. The List ADT has the following operations,

1. Insert Ascending

2. Merge

3. Display

4. Exit

Option 1 inserts a node so the list is always in ascending order. Option 2 takes two lists as input, and merges two lists into a third list. The third list should also be in ascending order. Convert the file into a header file and include it in a C++ file. The second C++ consists of 3 lists and has the following operations,

```
1. Insert List1

2. Insert List2

3. Merge into List3

4. Display

5. Exit

*/


#include<stdio.h>

#include<stdlib.h>


class Link {

    struct Node{

        int data;

        struct Node * next;

    };


    struct Node *head;

    struct Node *gethead() {

        return head;

    }


    public:

        Link() {

            head = NULL;

        }


    void display();

    int insertascending(int);
```

```
    int merge(Link,Link);
};

//Method to display the singely linked list.
void Link::display()
{
    struct Node* temp;
    temp = head;

    while(temp!=NULL) {
    printf("%d ",temp->data);
    temp = temp->next;
    }

}

//Method to insert in ascending order in the singely linked list.
int Link::insertascending(int num) {
    struct Node *newnode = (struct Node*)malloc(sizeof(struct Node));
    newnode->data = num;
    newnode->next = NULL;

    if (head == NULL || num <= head->data) {
      newnode->next = head;
      head=newnode;
    }

    struct Node *prev = head;
```

```
    struct Node *curr = head->next;

    while (curr != NULL && num > curr->data) {
        prev = curr;
        curr = curr->next;
    }

    prev->next = newnode;
    newnode->next = curr;

    return 1;
}

//Method to merge two singely linked list.
int Link::merge(Link l1,Link l2) {
    struct Node * temp1;
    struct Node * temp2;
    temp1 = l1.gethead();

    while(temp1!=NULL) {
      insertascending(temp1->data);
      temp1 = temp1->next;
    }

    temp2 = l2.gethead();
    while(temp2!=NULL) {
        insertascending(temp2->data);
        temp2 = temp2->next;
```

```
    }
    return 1;

}
```

CPP FILE:

```
/*
B. Write a C++ menu-driven program to implement List ADT using a singly linked list.
You have a gethead() private member function that returns the address of the head value
of a list. Maintain proper boundary conditions and follow good coding practices. The List
ADT has the following operations,

1. Insert Ascending
2. Merge
3. Display
4. Exit

Option 1 inserts a node so the list is always in ascending order. Option 2 takes two lists
as input, and merges two lists into a third list. The third list should also be in ascending
order. Convert the file into a header file and include it in a C++ file. The second C++
consists of 3 lists and has the following operations,

1. Insert List1
2. Insert List2
3. Merge into List3
4. Display
5. Exit
*/
#include<stdio.h>
#include<stdlib.h>
```

```
#include"insertion.h"


int main() {
    Link l1,l2,l3;
    int n1,n2,option;


    while(1) {
        printf("\n Enter the option\n1.INSERT NUMBER IN LIST 1\n2.INSERT NUMBER IN
LIST 2\n3.MERGE LISTS\n4.DISPLAY\n5.EXIT \n");
        scanf("%d",&option);
        switch(option) {


            case 1:
                printf("\n Enter the number to enter");
                scanf("%d",&n1);


                l1.insertascending(n1);
                printf("\n Inserted.");
                break;


            case 2:
                printf("\n Enter the number to enter");
                scanf("%d",&n2);


                l2.insertascending(n2);
                printf("\n Inserted");
                break;


            case 3:
```

```
            l3.merge(l1,l2);
            printf("\n Merged both list.");
            break;

        case 4:
            printf("\n LIST 1: ");
            l1.display();

            printf("\n LIST 2: ");
            l2.display();

            printf("\nLIST 3: ");
            l3.display();
            break;

        case 5:
            exit(0);
            break;

        default:
            printf("\n Enter a valid option.");
            break;

    }
    }

}
```

OUTPUT:

Q4 b

```
 Enter the option
1.INSERT NUMBER IN LIST 1
2.INSERT NUMBER IN LIST 2
3.MERGE LISTS
4.DISPLAY
5.EXIT
1

 Enter the number to enter1

 Inserted.
 Enter the option
1.INSERT NUMBER IN LIST 1
2.INSERT NUMBER IN LIST 2
3.MERGE LISTS
4.DISPLAY
5.EXIT
1

 Enter the number to enter2

 Inserted.
```

```
 Enter the option
1.INSERT NUMBER IN LIST 1
2.INSERT NUMBER IN LIST 2
3.MERGE LISTS
4.DISPLAY
5.EXIT
2

 Enter the number to enter3

 Inserted
 Enter the option
1.INSERT NUMBER IN LIST 1
2.INSERT NUMBER IN LIST 2
3.MERGE LISTS
4.DISPLAY
5.EXIT
2

 Enter the number to enter4

 Inserted
```

```
 Enter the option
1.INSERT NUMBER IN LIST 1
2.INSERT NUMBER IN LIST 2
3.MERGE LISTS
4.DISPLAY
5.EXIT
3

 Merged both list.
 Enter the option
1.INSERT NUMBER IN LIST 1
2.INSERT NUMBER IN LIST 2
3.MERGE LISTS
4.DISPLAY
5.EXIT
4

 LIST 1: 1 2
 LIST 2: 3 4
LIST 3: 1 2 3 4
 Enter the option
1.INSERT NUMBER IN LIST 1
2.INSERT NUMBER IN LIST 2
3.MERGE LISTS
4.DISPLAY
5.EXIT
5
```

WEEK 5-LIST ADT- DOUBLY LINKED LIST **Date-21/02/24**

AIM:

To Write a C++ menu-driven program to implement List ADT using a doubly linked list

TIME COMPLEXITY ANALYSIS:

1.Insert at beginning : O(1)
2.Insert at end : O(n)
3.Insert at position: O(n)
4.Delete at beginning : O(1)
5.Delete at end : O(n)
6.Delete at position : O(n)
7.Search : O(n)
8.Display : O(n)

ALGORITHM

1.INSERT AT BEGINNING:

Input:num,head
Output: 0 or 1

```
 Initialize newnode
 if head=NULL
 Head=newnode
 Return 1
Else
 Head->prev=newnode
 Newnode->next=head
 Head=newnode
 Return 1
```

2.INSERT AT END:

Input : num,head
Output : 0 or 1

```
If head=NULL
  Head=newnode
  Return 1
Else
  Set temp=head
  Repeat until temp->next!=NULL
     Temp=temp->next
  Newnode->prev=temp
  Temp->next=newnode
  Return 1
```

3.INSERT AT POS:

Input : num,pos,head
Output : 0 or 1

```
If pos=0
  Call insertbeg
Else
  Set temp=head
  Repeat until temp->next!=NULL and pos!=1
     Temp2=temp
     Temp=temp->next
     Pos- -
  Temp2->next=newnode
  Newnode->prev=temp2
  Newnode->next=temp
  Return 1
```

4.DELETE AT BEGINNING:

Input : head
Output : 0 or 1

```
If head=NULL
  Return 0
Else
  Head=head->next
  Head->prev=NULL
  Return 1
```

5.DELETE AT END:

Input : head
Output : 0 or 1

```
If head=NULL
  Return 0
Else
  Set temp=head
  Repeat until temp->next!=NULL
     Temp=temp->next
  Temp2=temp->next
  Temp->next=NULL
  Free temp2
  Return 1
```

6.DELETE AT POS:

Input : head
Output : 0 or 1

```
If head==NULL
  Return 0
Else if pos=1
  Call delbeg
Else
   Set temp=head
   Repeat until temp->next!=NULL and pos!=1
      Temp2=temp
      Temp=temp->next
   Temp2->next=temp->next
   Temp->prev=temp2
   Free temp1
```

7.SEARCH:

Input: head
Output: 0 or 1

```
If head=NULL
  Return 0
Else
   Set Temp=head
   Repeat until temp->next!=NULL
      If temp->data=num
        Return 1
      Temp=temp->next
   Return 0
```

8.DISPLAY:

Input:head
Output: All the elements in the list

```
If head=NULL
  Display the list is empty
Else
  Temp=head
  Repeat until temp!=NULL
     Display temp->data
     Temp=temp->next
```

CODE:

/*

A. Write a C++ menu-driven program to implement List ADT using a doubly linked list. Maintain proper boundary conditions and follow good coding practices. The List ADT has the following operations,

```
1. Insert Beginning

2. Insert End

3. Insert Position

4. Delete Beginning

5. Delete End

6. Delete Position

7. Search

8. Display

9. Exit


What is the time complexity of each of the operations?

*/


#include <stdio.h>

#include <stdlib.h>


class Dlist {

   struct Node {

      int data;

      struct Node *next;

      struct Node *previous;

   };


   struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));

   struct Node *head;

   struct Node *tail;


   public:

      Dlist() {

         head = NULL;

         tail = NULL;
```

```
    }
    void insertbeg(int num);
    void insertend(int num);
    int insertposition(int num ,int pos);
    void deletebeg();
    void deleteend();
    void deletepos(int pos);
    int size();
    int search(int num);
    void display();
    void displayreverse();
};

int main() {
  Dlist l1;
  int choice;
  int num;
  int pos;
  while(1) {

    printf("DOUBLEY LINKED LIST \n");
    printf("\n 1. Insert Beginning \n 2. Insert End \n 3. Insert Position");
    printf("\n 4. Delete Beginning \n 5. Delete End \n 6. Delete Position");
    printf("\n 7. Search \n 8. Display \n 9.Exit");
    printf("\n Enter the choice ");
    scanf("%d",&choice);

  switch(choice) {
    case 1:
      printf("Enter the number ");
      scanf("%d",&num);
```

```
  l1.insertbeg(num);
  break;


case 2:
  printf("Enter the number ");
  scanf("%d",&num);
  l1.insertend(num);
  break;

case 3:
  printf("Enter the number and the position ");
  scanf("%d %d",&num,&pos);
  if(l1.insertposition(num,pos)) {
    printf("\nInserted.");
  }
  else {
    printf("\nCannot insert.");
  }
  break;

case 4:
  l1.deletebeg();
  break;

case 5:
  l1.deleteend();
  break;

case 6:
  printf("Enter the position to delete ");
```

```
        scanf("%d",&pos);
        l1.deletepos(pos);
        break;

      case 7:
        printf("Enter the number to search ");
        scanf("%d",&num);
        num = l1.search(num);
        printf("The number is in %d pos \n",num);
        break;
      case 8:
        l1.display();
        break;

      case 9:
        printf("PROGRAM ENDED\n");
        return 0;



    }


  }
}

//Getting the size of the doubly linked list.
int Dlist::size() {
  struct Node *temp = head;
  int count = 0;
  if(head == NULL) {
    return 0;
  }
```

```
    else {
        while(temp != NULL) {
            count = count + 1;
            temp = temp -> next;
        }
        return count;
    }
}

//Inserting at the beginning of the doubley linked list.
//Time complexity => O(1)
void Dlist::insertbeg(int num) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));

    if(head == NULL) {
        newnode -> data = num;
        newnode -> next = head;
        newnode -> previous = NULL;
        head = newnode;
        tail = newnode;
        printf("Inserted %d successfully\n",num);
    }

    else {
        newnode -> data = num;
        newnode -> previous = NULL;
        newnode -> next = head;
        head -> previous = newnode;
        head = newnode;
        printf("Inserted %d successfully\n",num);
```

```
    }



}

//Inserting at the end of the doubley linked list.
//Time complexity => O(1)
void Dlist::insertend(int num) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));

    if(head == NULL) {
        insertbeg(num);
    }

    else {
        newnode -> data = num;
        tail -> next = newnode;
        newnode -> previous = tail;
        newnode -> next = NULL;
        tail = newnode;
        printf("Inserted %d successfully\n",num);
    }
}

//Inserting at the position of the doubley linked list.
//Time complexity => O(n)
int Dlist::insertposition(int val, int pos) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    if (pos == 0 || head == nullptr) {
        insertbeg(val);
    }
```

```
if (newnode == nullptr) {
    return 0;
}

newnode->data = val;

struct Node* temp = head;

for (int i = 1; i < pos; i++) {

    if (temp == nullptr) {
        return 0;
    }

    temp = temp->next;

}

if (temp == nullptr) {
    return 0;
}

newnode->next = temp->next;
newnode->previous = temp;

if (temp->next != nullptr) {
    temp->next->previous = newnode;
}
temp->next = newnode;
printf("Inserted %d successfully\n",val);
```

```
    return 1;
}

//Deleting the node at the beginning of the doubley linked list.
//Time complexity => O(1)
void Dlist::deletebeg() {

    if(head == NULL) {
        printf("The list is empty.");
    }

    else if(head -> next == NULL) {
        printf("Deleted %d successfully\n",head->data);
        head = NULL;

    }

    else {
        printf("Deleted %d successfully\n",head->data);
        head = head -> next;
        head -> previous = NULL;
    }
}

//Deleting the node at the end of the doubley linked list.
//Time complexity => O(1)
void Dlist::deleteend() {
    struct Node *temp = tail->previous;

    if(head == NULL) {
        printf("The list is empty.");
```

```
    }

    else if(head == tail) {
        deletebeg();
    }

    else {
        printf("Deleted %d successfully\n",temp->next->data);
        temp -> next = NULL;
        tail = temp;

    }
}

//Deleting the node at the given position in the doubley linked list.
//Time complexity => O(n)
void Dlist::deletepos(int pos) {
    int count = 0;
    struct Node *temp = head;
    struct Node *temp2;

    if(pos == 0) {
        deletebeg();
    }

    else if(pos == size() - 1) {
        deleteend();
    }

    else {
        while(count < pos -1) {
```

```
        temp = temp -> next;
        count = count + 1;
      }

      if(temp -> next != NULL) {
        temp2 = temp -> next -> next;
        temp -> next = temp2;
        temp2 -> previous = temp;

      }

      else {
        deleteend();
      }

  }
}

//Search for the value in the doubly linked list.
//Time complexity => O(n)
int Dlist::search(int num) {
  if (head == nullptr) {
    return 0;
  }
  struct Node* temp = head;
  int pos = 0;
  while (temp != nullptr && temp->data != num) {
    temp = temp->next;
    pos++;
  }
  if (temp == nullptr) {
```

```
        return 0;
    }
    return pos+1;
}

//Displaying the data of each nodes in the doubley linked list.
//Time complexity => O(n)
void Dlist::display() {
    struct Node *temp = head;

    if(head == NULL) {
        printf("List empty.");
    }

    else {
        while(temp != NULL) {
            printf("%d\n",temp -> data);
            temp = temp -> next;
        }
    }

}
```

OUTPUT:

```
DOUBLEY LINKED LIST

 1. Insert Beginning
 2. Insert End
 3. Insert Position
 4. Delete Beginning
 5. Delete End
 6. Delete Position
 7. Search
 8. Display
 9.Exit
 Enter the choice 1
Enter the number 4
Inserted 4 successfully
```

```
 Enter the choice 1
Enter the number 3
Inserted 3 successfully
```

```
 Enter the choice 1
Enter the number 2
Inserted 2 successfully
```

```
 Enter the choice 1
Enter the number 1
Inserted 1 successfully
```

```
 Enter the choice 8
1
2
3
4
```

```
DOUBLEY LINKED LIST

 1. Insert Beginning
 2. Insert End
 3. Insert Position
 4. Delete Beginning
 5. Delete End
 6. Delete Position
 7. Search
 8. Display
 9.Exit
 Enter the choice 2
Enter the number 5
Inserted 5 successfully
```

```
  Enter the choice 8
 1
 2
 3
 4
 5
```

```
  Enter the choice 3
 Enter the number and the position 6 5
```

```
  Enter the choice 8
 1
 2
 3
 4
 5
 6
```

```
  Enter the choice 5
 Deleted 6 successfully
```

```
 Enter the choice 8
1
2
3
4
5
```

```
 Enter the choice 6
Enter the position to delete 4
Deleted 5 successfully
```

```
 Enter the choice 8
1
2
3
4
```

```
 Enter the choice 7
Enter the number to search 2
The number is in 2 pos
```

```
 Enter the choice 8
1
2
3
4
```

```
 Enter the choice 9
PROGRAM ENDED
```

2)

AIM:

Write a C++ menu-driven program to implement a browser's front and back functionality.

ALGORITHM:

1.INSERT WEBPAGE:

Input: webpage,head ,current

Output : 0 or 1

```
If head=NULL
   Head=newnode
   Current=newode
Else if head!=current
   Set temp1=head
   Repeat until head!=current
       Head=head->next
       Temp2=temp1
       Temp1=temp1->next
       Free temp2
   Return insert(webpage)
Else
    Head->prev=newnode
    Newnode->next=newnode
    Head=newnode
    Return 1
```

2.FRONT:

Input: head,current

Output : 0 or 1

TIME COMPLEXITY:

CODE:

OUTPUT:

```
Enter the option
 1 - New Webpage
 2 - Go Front
 3 - Go Back
 4 - Exit
Enter the option 1
```

```
Inserted new page.0
(Press any key to continue)
```

```
Enter the option
 1 - New Webpage
 2 - Go Front
 3 - Go Back
 4 - Exit
Enter the option 1
```

```
Inserted new page.1
(Press any key to continue)
```

```
Enter the option
 1 - New Webpage
 2 - Go Front
 3 - Go Back
 4 - Exit
Enter the option 3
```

```
Moved back.
Current page: 0
(Press any key to continue)
```

```
Enter the option
 1 - New Webpage
 2 - Go Front
 3 - Go Back
 4 - Exit
Enter the option 2
```

```
Moved front.
Current page: 1
(Press any key to continue)
```

```
Enter the option
 1 - New Webpage
 2 - Go Front
 3 - Go Back
 4 - Exit
Enter the option 4
```

```
PROGRAM ENDED
PS C:\Users\kesav\OneDrive - SSN Trust\YEAR1_DStruct\CODE\LAB\LAB5(21-02-20
24)(DOUBLYLINKEDLIST)\QB>
```

WEEK 6- STACK ADT

Date-28/02/24

AIM:

To Write a separate C++ menu-driven program to implement stack ADT using a character array of size 5

ALGORITHM:

1.PUSH:

Input: chr

Output: 1 if element is pushed, 0 otherwise

If top equals size-1

    Return 0 indicating stack is full

Else

    Increment top

    Assign chr to arr at index top

    Return 1 indicating success

End If

2.POP:

Output: 1 if element is popped, 0 otherwise

If top equals -1

    Return 0 indicating stack is empty

Else

    For i from 0 to size-1

        Assign value of arr[i+1] to arr[i]

    End For

    Decrement top

Return 1 indicating success

End If

3.PEEK:

Output: Print the top element of the stack

If top is not equal to -1

Print arr at index top

Else

Print message indicating stack is empty

End If

TIME COMPLEXITY:

1. PUSH: O(1)

2. POP: O(1)

3. PEEK: O(1)

CODE:

/*

A. Write a separate C++ menu-driven program to implement stack ADT using a character array of size 5 and a singly linked list. Maintain proper boundary conditions and follow good coding practices. The Stack ADT has the following operations,

1. Push

2. Pop

3. Peek

4. Exit

What is the time complexity of each of the operations? (K4)

*/

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
class stack
{
   char arr[SIZE];
   int top;
   public:
      stack() {
         top = -1;
      }
      int push(char);
      int pop();
      char peak();
      void display();
};

int main() {
   stack l1;
   int choice;
   char element;
   int pos;
   while(1) {

      printf("\n STACK ADT \n");
      printf("\n 1. Push \n 2. Pop \n 3. Peak \n 4. Exit \n");
      scanf("%d",&choice);
```

```
switch(choice) {
  case 1:
    printf("Enter the character ");
    scanf(" %c",&element);
    if(l1.push(element)) {
      printf("\n Inserted successfully.\n");
    }
    else {
      printf("\n Insertion unsuccessful.\n");
    }
    break;

  case 2:
    if(l1.pop()) {
      printf("\n Succesful.\n");
    }
    else {
      printf("\n Stack empty.\n");
    }
    break;

  case 3:
    if(!l1.peak()) {
      printf("\n Stack is empty.\n");
    }

    else {
      printf("\n %c\n",l1.peak());
    }
```

```
            break;

          case 4:
            return 0;

      }
   }

}

//Method to push into stack adt.
//Time complexity => O(1)
int stack::push(char element) {
   if(top == SIZE-1) {
      return 0;
   }
   else if (top == -1) {
      top = 0;
      arr[0] = element;
      return 1;
   }
   else {
      top = top + 1;
      arr[top] = element;
      return 1;

   }
}

//Method to pop the element in stack adt.
```

```
//Time complexity => O(1)
int stack::pop() {
   if(top == -1) {
      return 0;
   }
   else {
      top = top - 1;
      return 1;
   }
}

//Displaying the top of the node.
//Time complexity => O(1)
char stack::peak() {
   if(top == -1) {
      return 0;
   }

   else {
      return arr[top];
   }
}

//Displaying the values of the stack adt.
//Time complexity => O(n)
void stack::display() {
   printf("\nThe contents of the stack are:");
   for(int i=top;i>=0;i--) {
      printf("%c ",arr[i]);
   }
```

}

OUTPUT:

```
 STACK ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
1
Enter the character 1

 Inserted successfully.

 STACK ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
1
Enter the character 2

 Inserted successfully.
```

```
 STACK ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
2

 Succesful.

 STACK ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
3

 1

 STACK ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
4
```

QA)2)

AIM:

To Write a separate C++ menu-driven program to implement stack ADT using a character singly linked list.

ALGORITHM:

1.PUSH:

Input: chr,stack

Output: 1 if element is pushed, 0 otherwise

Create a new node with data set to chr

If head is NULL

    Set newnode next to NULL

    Set head to newnode

Else

    Set newnode next to head

    Set head to newnode

End If

Return 1 indicating success

2.POP:

Output: 1 if element is popped, 0 otherwise

If head is NULL

    Return 0 indicating stack is empty

Else

    Set temp to head

Set head to head next

Free temp

Return 1 indicating success

End If

3.PEEK:

Check if the head of the stack is NULL, indicating the stack is empty.

If the stack is empty, print "The stack is empty".

Otherwise, print the data of the head node, which is the top element of the stack.

TIME COMPLEXITY:

1.PUSH:O(1)

2.POP:O(1)

3.PEEK:O(1)

CODE:

```
#include <stdio.h>
#include <stdlib.h>

class list {
   struct Node {
      int data;
      struct Node *next;
   };

   struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
   struct Node *head;
```

```
    public:
        list() {
            head = NULL;
        }
        int push(char);
        int pop();
        void peak();
        void display();


};

int main() {
    list l1;
    int choice;
    char element;
    int pos;
    while(1) {

        printf("\n list ADT \n");
        printf("\n 1. Push \n 2. Pop \n 3. Peak \n 4. Exit \n");
        scanf("%d",&choice);

        switch(choice) {
            case 1:
                printf("Enter the character ");
                scanf(" %c",&element);
                if(l1.push(element)) {
                    printf("\n Inserted successfully.\n");
                }
```

```
            else {
               printf("\n Insertion unsuccessful.\n");
            }
            break;

         case 2:
            if(l1.pop()) {
               printf("\n Succesful.\n");
            }
            else {
               printf("\n Stack empty.\n");
            }
            break;

         case 3:
            l1.peak();
            break;

         case 4:
            return 1;

      }
   }

}

//Pushing the element into the singly linked list.
//Time complexity => O(1)
int list::push(char element) {
   struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
```

```
    newnode -> data = element;
    newnode -> next = head;
    head = newnode;
    return 1;

}

//Popping the element from the singly linked list.
//Time complexity => O(1)
int list::pop() {
    struct Node *temp;
    struct Node *temp2;
    if(head == NULL) {
        printf("The list is empty.");
        return 0;
    }

    else {
        temp2 = head; //temp2 stores the memory address of 0 element
        temp = head -> next; //temp stores the memory of 1 element
        head = temp;
        free(temp2);
        return 1;
    }
}

//Displaying the top of the singly linked list.
//Time complexity => O(1)
void list::peak() {
    if(head == NULL) {
```

```
        printf("Stack is empty.\n");
    }
    else {
        printf("%c",head->data);
    }
}

//Displaying the data of each nodes in singly linked list.
//Time complexity => O(n)
void list::display() {
    struct Node *temp = head;
    if(head == NULL) {
        printf("list empty");
    }
    else {
        while(temp != NULL) {
            printf("\n %c \n",temp->data);
            temp=temp->next;
        }
    }
}
```

OUTPUT:

```
 list ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
1
Enter the character a

 Inserted successfully.

 list ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
1
Enter the character b

 Inserted successfully.
```

```
 list ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
2

 Succesful.

 list ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
3
a
 list ADT

 1. Push
 2. Pop
 3. Peak
 4. Exit
4
```

QB)

AIM:

To Write a C++ menu-driven program to implement infix to postfix and postfix evaluation.

ALGORITHM:

Stack headerfile is imported from the previous code

CHECK:

Input: arr (an array of characters), I1 (pointer to operator stack), I2 (pointer to output stack)

Output: stack - The stacks are modified directly

1. Initialize count to 0
2. Loop through each character in arr until a null terminator '\0' is encountered:
   a. If the current character is '*', '/' or '%':
      i. If count is 3:
         A. Push the top element of I1 onto I2
         B. Pop the top element from I1
         C. If I1 is empty, set count to 0
         D. Otherwise, update count based on the new top of I1
      ii. Else:
         A. Push the current character onto I1
         B. Set count to 3
   b. If the current character is '+' or '-':
      i. If count is 3 or 2:
         A. Repeat steps i.A to i.D as above
      ii. Else:
         A. Push the current character onto I1
         B. Set count to 2
   c. If the current character is '=':
      i. If count is 1:
         A. Repeat steps i.A to i.D as above

ii. Else:

A. Push the current character onto l1

B. Set count to 1

d. If the current character is an operand (not an operator or '='):

i. Push the current character onto l2

3. While l1 is not empty:

a. Push the top element of l1 onto l2

b. Pop the top element from l1

End Loop

TIME COMPLEXITY:

<u>Check:O(n)</u>

CODE:

Header file:

/*

C. Write a C++ menu-driven program to get a string of '(' and ')' parenthesis from the user and check whether they are balanced. Identify the optimal ADT and data structure to solve the mentioned problem. You can consider all previous header files for the solution's implementation. Maintain proper boundary conditions and follow good coding practices. The program has the following operations,

1. Check Balance

2. Exit

The Check Balance operations get a string of open and closed parentheses. Additionally, it displays whether the parenthesis is balanced or not. Explore at least two designs (solutions) before implementing your solution.

What is the time complexity of each solution, and what is the optimal solution? Justify your answer.

```
*/

#include<stdio.h>
#include<stdlib.h>
#include <string.h>
#include <ctype.h>

class list{
   private:
      struct node{
         char data;
         struct node* prev;
      };
      struct node* top;
   public:
      list(){
         top=NULL;
      }
      int push(char);
      char pop(void);
      char peek(void);
      void display(void);
};

//Push elements to the stack.
//Time complexity => O(1).
int list::push(char chr){
   struct node* newnode = (struct node*)malloc(sizeof(struct node));
   if(newnode==NULL){
      return 0;
```

```
    }
    else{
        newnode->data=chr;
        newnode->prev=top;
        top=newnode;
        return 1;
    }
}

//Pop elements from the stack.
//Time complexity => O(1).
char list::pop(void){
    if(top==NULL){
        return '\0';
    }
    else{
        char res = top->data;
        struct node* copy = top;
        top=top->prev;
        free(copy);
        return res;
    }
}

//Display elements in the stack.
//Time complexity => O(n).
void list::display(void){
    if (top==NULL){
        printf("List is empty!");
    }
```

```
    else{
        struct node* temp = top;
        while (temp!=NULL){
            printf("%c ",temp->data);
            temp=temp->prev;
        }
    }
}

//Peak of the stack.
//Time complexity => O(1)
char list::peek(void){
    if(top==NULL){
        return '\0';
    }
    else{
        return top->data;
    }
}
```

CPP file:

```
#include "QB.h"

int main(){
    int precedence(char);
    while(1){
        list l;
        int val;
        int choice;
        int len;
```

```
    char infix[20];
    char postfix[20];
    printf("\nMenu:\n1. Get Infix\n2. Convert Infix\n3. Evaluate Postfix\n4.
Exit\nEnter your choice: ");
    scanf("%d",&choice);
    char chk = '0';
    int i;
    int len1=0;
    switch(choice){
      //Checking if the infix expression is valid.
      //Time complexity => O(n).
      case 1:{
        while(chk=='0'){
          len = 0;
          int oc = 0;
          int ec = 0;
          char let;
          printf("Enter a valid infix string : ");
          scanf("%s",&infix);
          let = infix[len];
          while(let!='\0'){
            if(len%2==0 && let!='+' && let!='-' && let!='*' && let!='/' &&
let!='%'){
              ec+=1;
            }
            else if(len%2!=0 && (let=='+' || let=='-' || let=='*' || let=='/' || let
=='%')){
              oc+=1;
            }
            len+=1;
            let = infix[len];
```

```
        }
        if(ec==len/2+1 && oc==len/2){
          printf("Valid infix expression");
          chk='1';
        }
        else{
          printf("Invalid infix expression\n");
        }
      }
      break;
    }
    //Convert infix to postfix.
    //Time complexity => O(n)
    case 2:
      for(i=0;i<len;i++){
        if (i%2==0){
          postfix[len1]=infix[i];
          len1+=1;
        }
        else{
          if(l.peek()=='\0'){
            l.push(infix[i]);
          }
          else if(precedence(infix[i]) > precedence(l.peek())){
            l.push(infix[i]);
          }
          else if(precedence(infix[i]) == precedence(l.peek())){
            if(infix[i]!='='){
              postfix[len1]=l.pop();
              l.push(infix[i]);
```

```
            len1+=1;
          }
          else{
            l.push(infix[i]);
          }
        }
        else{
          while(precedence(infix[i]) <= precedence(l.peek())) {
            postfix[len1] = l.pop();
            len1 += 1;
          }
          l.push(infix[i]);
        }
      }
    }
    while (l.peek() != '\0') {
      postfix[len1] = l.pop();
      len1 += 1;
    }
    postfix[len1]='\0';
    printf("%s\n",postfix);
    break;
  //Evaluate postfix expression
  //Time complexity => O(n)
  case 3:
    for(i=0;i<len;i++){
      if(isalpha(postfix[i])!=0){
        printf("Enter the value of %c: ",postfix[i]);
        scanf("%d",&val);
        l.push(char(val));
```

```
            }
            else{
                switch(postfix[i]){
                    case '+':
                        l.push(char(int(l.pop())+int(l.pop())));
                        break;
                    case '-':
                        l.push(char(int(l.pop())-int(l.pop())));
                        break;
                    case '*':
                        l.push(char(int(l.pop())*int(l.pop())));
                        break;
                    case '/':
                        l.push(char(int(l.pop())/int(l.pop())));
                        break;
                    case '%':
                        l.push(char(int(l.pop())%int(l.pop())));
                        break;
                }
            }
        }
        printf("\nAnswer %d",int(l.pop()));
        break;
    case 4:
        printf("PROGRAM ENDED\n");
        return 0;
    default:
        printf("Enter a valid choice.");
        break;
}
```

```
    }
}

//Function to check the precedence of operators.
int precedence(char op) {
    if(op=='*' || op=='/' || op=='%')
        return 2;
    else if (op=='+' || op=='-')
        return 1;
    else
        return 0;
}
```

OUTPUT:

```
Menu:
1. Get Infix
2. Convert Infix
3. Evaluate Postfix
4. Exit
Enter your choice: 1
Enter a valid infix string : a+b-c+d
Valid infix expression
Menu:
1. Get Infix
2. Convert Infix
3. Evaluate Postfix
4. Exit
Enter your choice: 2
ab+c-d+

Menu:
1. Get Infix
2. Convert Infix
3. Evaluate Postfix
4. Exit
Enter your choice: 3
Enter the value of a: 1
Enter the value of b: 2
Enter the value of c: 3
Enter the value of d: 4

Answer 4
Menu:
1. Get Infix
2. Convert Infix
3. Evaluate Postfix
4. Exit
Enter your choice: 4
PROGRAM ENDED
```

QC)

AIM:

To Write a C++ menu-driven program to get a string of '(' and ')' parenthesis from the user and check whether they are balanced.

ALGORITHM:

Stack headerfile is imported from the previous code

Check:

Input: arr (an array of characters)

Output: 1 if parentheses are balanced, 0 otherwise

1. Loop through each character in arr until a null terminator '\0' is encountered:
   a. If the current character is '(':
      i. Push 'a' onto the stack.
   b. If the current character is ')':
      i. If the stack is empty (head is NULL), return 0 (unbalanced).
      ii. Otherwise, pop the top element from the stack.

End Loop

2. After the loop, check if the stack is empty:
   a. If the stack is empty (head is NULL), return 1 (parentheses are balanced).
   b. Otherwise, return 0 (parentheses are not balanced).

TIME COMPLEXITY:

CHECK:O(N)

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define MAX_SIZE 100

class stack {
private:
   struct Node {
      char data;
      struct Node* next;
   };

   struct Node* top;

public:
   stack();
   bool isEmpty();
   int push(char item);
   int pop();
   int getSize();
   bool isBalanced(const char* str);
   void clear();
};

int main() {
   stack stack;
   char input[MAX_SIZE];
```

```
    int choice;
    while(1) {
        printf("1. Check Balance\n");
        printf("2. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar();

        switch (choice) {
            case 1:
                printf("Enter a string of parentheses: ");
                fgets(input, MAX_SIZE, stdin);
                input[strcspn(input, "\n")] = '\0';
                if (stack.isBalanced(input)) {
                    printf("Parentheses are balanced.\n");
                } else {
                    printf("Parentheses are not balanced.\n");
                }
                stack.clear();
                break;
            case 2:
                printf("PROGRAM ENDED\n");
                return 0;
            default:
                printf("Invalid.\n");
        }
    }
    return 0;
}
```

```
stack::stack() {
   top = NULL;
}

bool stack::isEmpty() {
   return top == NULL;
}

//Pushing each character in the string in the stack.
//Time complexity => O(1).
int stack::push(char item) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed\n");
      return 0;
   }
   newNode->data = item;
   newNode->next = top;
   top = newNode;
   return 1;
}

//Popping each character in the string in the stack.
//Time complexity => O(1).
int stack::pop() {
   if(isEmpty()) {
      printf("Underflow\n");
      return 0;
   }
   struct Node* temp = top;
```

```
    top = temp->next;
    free(temp);
    return 1;
}

//Getting the size of the stack.
//Time complexity => O(n).
int stack::getSize() {
    int size = 0;
    struct Node* current = top;
    while (current != NULL) {
        size++;
        current = current->next;
    }
    return size;
}

//Checking if the parenthesis is balanced.
//Time complexity => O(n)
bool stack::isBalanced(const char* str) {
    for (int i = 0; str[i]; i++) {
        if (str[i] == '(') {
            push('(');
        } else if (str[i] == ')') {
            if (isEmpty() || pop() != 1) {
                return false;
            }
        }
    }
```

```
    bool result = isEmpty();
    return result;
}

//Clearing the data from the stack.
//Time complexity => O(n)
void stack::clear() {
    while (!isEmpty()) {
        pop();
    }
}
```

OUTPUT:

```
1. Check Balance
2. Exit
Enter your choice: 1
Enter a string of parentheses: ((((((((())
Parentheses are not balanced.
1. Check Balance
2. Exit
Enter your choice: 1
Enter a string of parentheses: (((((())))))()
Parentheses are balanced.
1. Check Balance
2. Exit
Enter your choice: 2
PROGRAM ENDED
```

WEEK 7-QUEUE AND CIRCULAR QUEUE ADT **Date:06/03/2024**

AIM:

T0 Write a separate C++ menu-driven program to implement Queue ADT using an integer array of size 5.

ALGORITHM:

1.ENQUEUE:

Input: num (an integer to be added to the queue),arr

Output: 1 if the operation is successful, 0 otherwise

1. Check if the queue is full:
   - If (rear + 1) % size == front, then return 0 (queue is full).
2. Check if the queue is empty:
   - If front == -1, then set front and rear to 0.
3. Otherwise:
   - Set rear to (rear + 1) % size.
   - Store num at arr[rear].
4. Return 1 indicating that the operation was successful.

2.DEQUEUE:

Input: arr

Output: 1 if the operation is successful, 0 otherwise

1. Check if the queue is empty:
   - If front == -1, then return 0 (queue is empty).
2. Check if the front equals the rear:
   - If true, then set front and rear to -1 (the queue will be empty after this operation).

3. Otherwise:

   - Increment front using front = (front + 1) % size.

4. Return 1 indicating that the operation was successful.

3.PEEK:

Output: Prints the element at the front of the queue

1. Check if the queue is empty:

   - If front == -1, print "The queue is empty".

2. Otherwise:

   - Print the element at arr[front].

TIME COMPLEXITY:

1. **Enqueue**: O(1)
2. **Dequeue**: O(1)
3. **Peek**: O(1)

CODE:

/*

A. Write a separate C++ menu-driven program to implement Queue ADT using an integer array of size 5. Maintain proper boundary conditions and follow good coding practices. The Queue ADT has the following operations,

1. Enqueue

2. Dequeue

3. Peek

4. Exit

What is the time complexity of each of the operations?

*/

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
class queue {
   int arr[SIZE];
   int front;
   int rear;
   public:
      queue() {
         front = -1;
         rear = -1;
      }
      int isfull();
      int isempty();
      int enqueue(int);
      int dequeue();
      void peek();
      void display();
};

int main() {
   queue l1;
   int choice;
   int element;
   int pos;
   while(1) {

      printf("\n Queue \n");
      printf("\n 1.Enqueue \n 2.Dequeue \n 3.Peek \n 4.Exit \n");
      scanf("%d",&choice);
```

```
switch(choice) {
   case 1:
      printf("Enter the numbers ");
      scanf(" %d",&element);
      if(l1.enqueue(element)) {
         printf("\n Inserted successfully.\n");
      }
      else {
         printf("\n Insertion unsuccessful.\n");
      }
      break;

   case 2:
      if(l1.dequeue()) {
         printf("\n Dequeue Succesful.\n");
      }
      else {
         printf("\n Queue empty.\n");
      }
      break;

   case 3:
      l1.peek();
      break;

   case 4:
      printf("PROGRAM ENDED\n");
      return 0;
```

```
    }
  }

}

//Function to check if the queue is full.
//Time complexity => O(1)
int queue::isfull() {
  if(rear == SIZE -1) {
    return 1;
  }
  else {
    return 0;
  }
}

//Function to check if the queue is empty.
//Time complexity => O(1)
int queue::isempty() {
  if(front == -1) {
    return 1;
  }
  else {
    return 0;
  }
}

//Function to add the element to the queue.
//Time complexity => O(1)
int queue::enqueue(int element) {
```

```
    if(isfull()) {
        return 0;
    }
    else {
        if(isempty()) {
            front = 0;
            rear = 0;
            arr[0] = element;
        }
        else {
            rear = rear + 1;
            arr[rear] = element;
        }
        return 1;

    }
}

//Function to pop the element from the queue.
//Time complexity => O(1)
int queue::dequeue() {
    if(isempty()) {
        return 0;
    }

    else {
        if(front == rear) {
            front = -1;
            rear = -1;
        }
```

```
        else {
            front = front + 1;
        }
        return 1;

    }
}

//Function to display the peek value in the queue.
//Time complexity => O(1)
void queue::peek() {
    if(isempty()) {
        printf("Stack empty.");
    }
    else {
        printf("The peek is %d\n",arr[front]);
    }

}

//Function to display the elements in the queue.
//Time complexity => O(n)
void queue::display() {
    if(isempty()) {
        printf("Stack empty.");
    }
    else {
        int temp2 = front;
        while(temp2 != rear + 1) {
            printf("The number %d\n",arr[temp2]);
```

```
            temp2 = temp2 + 1;
        }
    }
}
```

OUTPUT:

```
 Queue

 1.Enqueue
 2.Dequeue
 3.Peek
 4.Exit
1
Enter the numbers 1

 Inserted successfully.

 Queue

 1.Enqueue
 2.Dequeue
 3.Peek
 4.Exit
1
Enter the numbers 2

 Inserted successfully.

 Queue

 1.Enqueue
 2.Dequeue
 3.Peek
 4.Exit
2

 Dequeue Succesful.
```

```
 Queue

 1.Enqueue
 2.Dequeue
 3.Peek
 4.Exit
3
The peek is 2

 Queue

 1.Enqueue
 2.Dequeue
 3.Peek
 4.Exit
4
PROGRAM ENDED
```

Q)7)B)

AIM:

To Write a separate C++ menu-driven program to implement Circular Queue ADT using an integer array of size 5.

ALGORITHM:

1.ENQUEUE:

Input: num (the number to be inserted into the queue),arr

Output: 1 if the operation is successful, 0 otherwise

1. Check if the queue is full:
   - If (rear + 1) % size == front, return 0 (indicating the queue is full).
2. Check if the queue is empty:
   - If rear == -1 (meaning front will also be -1), set front and rear to 0.
3. Otherwise, adjust rear for circular behavior:
   - Set rear = (rear + 1) % size.
4. Place num at the position indicated by rear in the array.
5. Return 1 (indicating the operation was successful).

2.DEQUEUE:

Input: arr

Output: 1 if the operation is successful, 0 otherwise

1. Check if the queue is empty:
   - If rear == -1, return 0 (indicating the queue is empty).
2. Check if this is the last element in the queue:
   - If front == rear, reset front and rear to -1 (indicating the queue is now empty).
3. Otherwise, adjust front for circular behavior:

- Set front = (front + 1) % size.

4. Return 1 (indicating the operation was successful).

3.PEEK:

Input: arr

output: Print the front element of the queue or a message if the queue is empty

1. Check if the queue is empty:
   - If front == -1, print "The queue is empty!".
2. Otherwise:
   - Print the element at the front index of the array.

TIME COMPLEXITY:

1.ENQUEUE:O(1)

2.DEQUEUE:O(1)

3.PEEK:O(1)

CODE:

/*

B. Write a separate C++ menu-driven program to implement Circular Queue ADT using an integer array of size 5. Maintain proper boundary conditions and follow good coding practices. The Circular Queue ADT has the following operations,

1. Enqueue

2. Dequeue

3. Peek

4. Exit

What is the time complexity of each of the operations?

*/

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
class queue {
   int arr[SIZE];
   int front;
   int rear;
   public:
      queue() {
         front = -1;
         rear = -1;
      }
      int isfull();
      int isempty();
      void enqueue(int);
      void dequeue();
      void peek();
      void display();
};

int main() {
   queue l1;
   int choice;
   int num;
   int pos;
   while(1) {
```

```
    printf("\n CIRCULAR QUEUE \n");
    printf("\n 1.Push \n 2.Pop \n 3.peek \n 4.Exit \n");
    scanf("%d",&choice);

    switch(choice) {
      case 1:
        printf("Enter the numbers ");
        scanf(" %d",&num);
        l1.enqueue(num);
        break;

      case 2:
        l1.dequeue();
        break;

      case 3:
        l1.peek();
        break;

      case 4:
        printf("PROGRAM ENDED\n");
        return 0;

      case 5:
        l1.display();

    }
  }
```

```
}

//Function to check if the queue is full.
//Time complexity => O(1)
int queue::isfull() {
   if((rear+1)%SIZE == front) {
      return 1;
   }
   else {
      return 0;
   }
}

//Function to check if the queue is empty.
//Time complexity => O(1)
int queue::isempty() {
   if(front == -1 && rear == -1) {
      return 1;
   }
   else {
      return 0;
   }
}

//Function to add the element to the queue.
//Time complexity => O(1)
void queue::enqueue(int num) {

   if(isempty()) {
      arr[rear+1]=num;
```

```
        front++;
        rear++;
        printf("Inserted Successfully");
    }

    else if(isfull()==0) {
        arr[(rear+1)%SIZE]=num;
        rear=(rear+1)%SIZE;
        printf("Inserted Successfully");
    }

    else {
        printf("Not Inserted");
    }
}

//Function to delete the element from the queue.
//Time complexity => O(1)
void queue::dequeue() {
    if (isempty()) {
        printf("Queue Empty.");
    }

    else if(rear==front) {
        int b=arr[front];
        rear=-1;
        front=-1;
        printf("DEQUEUED");
    }
```

```
    else {
        int b=arr[front];
        front=(front+1)%SIZE;
        printf("DEQUEUED");
    }

}

//Function to print the peek of the queue.
//Time complexity => O(1)
void queue::peek() {
    if(isempty()) {
        printf("Queue Empty.");
    }

    else {
        printf("The peek is %d",arr[front]);
    }
}

//Function to display the elements in the queue.
//Time complexity => O(n)
void queue::display() {
    int i=front;
    if(isempty()==0)
    {
        while(i!=rear)
        {
            printf("%d,",arr[i]);
            i=(i+1)%SIZE;
```

```c
        }
        printf("%d",arr[i]);
    }
    else
    {
        printf("Queue Empty.");
    }
}
```

OUTPUT:

```
 CIRCULAR QUEUE

 1.Push
 2.Pop
 3.peek
 4.Exit
1
Enter the numbers 4
Inserted Successfully
 CIRCULAR QUEUE

 1.Push
 2.Pop
 3.peek
 4.Exit
1
Enter the numbers 5
Inserted Successfully
 CIRCULAR QUEUE

 1.Push
 2.Pop
 3.peek
 4.Exit
2
DEQUEUED
```

```
 CIRCULAR QUEUE

 1.Push
 2.Pop
 3.peek
 4.Exit
3
The peek is 5
 CIRCULAR QUEUE

 1.Push
 2.Pop
 3.peek
 4.Exit
4
PROGRAM ENDED
```

Q)C)

AIM:

To Write a separate C++ menu-driven program to implement Queue ADT using an integer-linked list

ALGORITHM:

1.ENQUEUE:

Input: num (integer value to be enqueued),list

Output: 1 if the enqueue operation is successful, 0 otherwise (though this setup always returns 1 upon successful memory allocation)

1. Create a new node dynamically allocating memory for it.
2. Set the data of the new node to num and its next pointer to NULL.
3. Check if the queue (front pointer) is empty:
   - If yes, set both front and rear pointers to this new node.
   - If no, append the new node to the end of the queue (rear->next) and move the rear pointer to this new node.
4. Return 1 to indicate that the enqueue operation was successful.

2.DEQUEUE:

Input: list

Output: 1 if an element is successfully dequeued, 0 if the queue is empty.

1. Check if the queue (front pointer) is empty:
   - If yes, return 0 indicating the queue is empty and nothing to dequeue.
2. Set a temporary pointer (temp) to the front.
3. Update the front pointer to the next node in the queue.

4. Free the memory of the node pointed by temp.

5. If after updating, the front becomes NULL, also set the rear to NULL (handling the last element removal).

6. Return 1 to indicate the dequeue operation was successful.

3.PEEK:

Input:list

Output: Print the value of the front element of the queue.

1. Check if the queue is empty (front is NULL):

   - If not empty, print the data value of the front node.

   - If empty, print an appropriate message indicating the queue is empty (though in your current code, this check is missing and could be added for safety).

TIME COMPLEXITY:

1.ENQUEUE:O(1)

2.DEQUEUE:O(1)

3.PEEK:O(1)

CODE:

/*

C. Write a separate C++ menu-driven program to implement Queue ADT using an integer-linked list. Maintain proper boundary conditions and follow good coding practices. The Queue ADT has the following operations,

1. Enqueue

2. Dequeue

3. Peek

4. Exit

What is the time complexity of each of the operations?
*/

```
#include <stdio.h>
#include <stdlib.h>

class List {
   struct Node {
      int data;
      struct Node *next;
   };

   struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
   struct Node *head;

   public:
      List() {
         head = NULL;
      }
      int insert_end(int num);
      void del_beg();
      void peek();
      void display();
      int size();
};

int main() {
   List l1;
   int choice;
```

```
int num;
int pos;
while(1) {

   printf("\n SINGELY LINKED LIST \n");
   printf("\n 1. Enqueue \n 2. Dequeue \n 3. peek \n 4. Exit \n");;
   printf("\n Enter the choice ");
   scanf("%d",&choice);

   switch(choice) {
      case 1:
         printf("Enter the number ");
         scanf("%d",&num);
         if(l1.insert_end(num)) {
            printf("\n Inserted successfully.");
         }
         else {
            printf("\n Insertion unsuccessful.");
         }
         break;

      case 2:
         l1.del_beg();
         break;

      case 3:
         l1.peek();
         break;

      case 4:
```

```
            printf("PROGRAM ENDED\n");
            return 1;
            break;

          case 5:
            l1.display();
            break;
        }
    }
}

//Function to add the element to the queue.
//Time complexity => O(n)
int List::insert_end(int num) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    struct Node *temp = head;
    newnode -> data = num;
    newnode -> next = NULL;
    if(head == NULL) {
        head = newnode;
        return 1;
    }

    else {
        while(temp -> next != NULL) {
            temp = temp -> next;

        }
        temp -> next = newnode;
        return 1;
```

```
    }

}

//Function to delete the element from the queue.
//Time complexity => O(1)
void List::del_beg() {
    struct Node *temp;
    struct Node *temp2;
    if(head == NULL) {
        printf("The list is empty.");
    }

    else {
        temp2 = head;
        printf("Deleted %d\n",temp2->data);
        temp = head -> next;
        head = temp;
        free(temp2);

    }
}

//Function to show the peek of queue
//Time complexity => O(1)
void List::peek() {
    if(head == NULL) {
        printf("Queue empty.");
    }
    else {
```

```
        printf("Peak is %d\n",head -> data);
    }
}

//Function to display the elements of the queue
//Time complexity => O(n)
void List::display() {
    struct Node *temp;
    temp = head;
    if(head == NULL) {
        printf("Queue empty.");
    }
    else {
        while(temp!= NULL) {
            printf("%d ",temp->data);
            temp = temp -> next;
        }
    }
}
```

OUTPUT:

```
 SINGELY LINKED LIST

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 1
Enter the number 4

 Inserted successfully.
 SINGELY LINKED LIST

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 1
Enter the number 5

 Inserted successfully.
```

```
 SINGELY LINKED LIST

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 2
Deleted 4

 SINGELY LINKED LIST

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 3
Peak is 5

 SINGELY LINKED LIST

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 4
PROGRAM ENDED
```

Q)D)

AIM:

To Write a separate C++ menu-driven program to implement Circular Queue ADT using an integer-linked list.

ALGORITHM:

1.ENQUEUE:

Input: num (integer to be added to the queue),list

Output: 1 on successful insertion, 0 on failure (though failure isn't possible unless memory allocation fails)

1. Allocate memory for a new node.
2. Set the data of the new node to num.
3. If the queue is empty:
   - Set the new node's next pointer to point to itself (making it circular).
   - Set both front and rear pointers to the new node.
4. If the queue is not empty:
   - Set the new node's next pointer to the front.
   - Update the rear's next pointer to point to the new node.
   - Move the rear pointer to the new node.
5. Return 1 to indicate success.

2.DEQUEUE:

Output: 1 if an element is successfully dequeued, 0 if the queue is empty.

1. If the queue is empty (front is NULL):

   - Return 0.

2. If the queue has only one node (front == rear):

   - Free the node.

   - Set front and rear to NULL.

3. If the queue has more than one node:

   - Set the front to the next node of the front.

   - Adjust the rear's next pointer to point to the new front.

   - Free the old front node.

4. Return 1 to indicate success.

3.PEEK:

Output: Displays the data of the front element.

1. If the queue is not empty:

   - Print the data of the front node.

2. If the queue is empty:

   - Print an appropriate message indicating that the queue is empty. (This check should ideally be included for safety)

TIME COMPLEXITY:

1.ENQUEUE:O(1)

2.DEQUEUE:O(1)

3.PEEK:O(1)

CODE:

```
/*
D. Write a separate C++ menu-driven program to implement Circular Queue ADT
using an integer-linked list. Maintain proper boundary conditions and follow
good coding practices. The Circular Queue ADT has the following operations,

1. Enqueue
2. Dequeue
3. Peek
4. Exit

What is the time complexity of each of the operations?
*/

#include <stdio.h>
#include <stdlib.h>

class List {
    struct Node {
        int data;
        struct Node *next;
    };

    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    struct Node *front;
    struct Node *rear;


    public:
        List() {
            front = NULL;
            rear = NULL;
        }
        int isempty();
        int insert_beg(int num);
        void delete_end();
        void peek();
        void display();



};

int main() {
    List l1;
    int choice;
    int num;
    int pos;
```

```
    while(1) {

        printf("\n CIRCULAR QUEUE \n");
        printf("\n 1. Enqueue \n 2. Dequeue \n 3. peek \n 4. Exit \n");;
        printf("\n Enter the choice ");
        scanf("%d",&choice);

        switch(choice) {
            case 1:
                printf("Enter the number ");
                scanf("%d",&num);
                if(l1.insert_beg(num)) {
                    printf("\n Inserted successfully.");
                }
                else {
                    printf("\n Insertion unsuccessful.");
                }
                break;

            case 2:
                l1.delete_end();
                break;

            case 3:
                l1.peek();
                break;

            case 4:
                printf("PROGRAM ENDED\n");
                return 1;
                break;

            case 5:
                l1.display();
                break;
        }
    }


}

int List::isempty() {
    if(front==NULL && rear==NULL) {
        return 1;
    }

    else {
        return 0;
    }
```

```
}

//Function to add the element to the queue.
//Time complexity => O(1)
int List::insert_beg(int num) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    newnode -> data = num;
    if(isempty()) {
        front=rear=newnode;
        rear->next=front;
        return 1;
    }

    else {
        rear->next=newnode;
        rear=newnode;
        rear->next=front;
        return 1;
    }
}

//Function to delete the element from the queue.
//Time complexity => O(1)
void List::delete_end() {
    struct Node *temp= front;
    if(isempty()) {
        printf("Queue empty.");
    }

    else if(front==rear) {
        int b=temp->data;
        printf("Deleted %d\n",b);
        front=rear=NULL;
        free(temp);
    }

    else {
        int b=temp->data;
        printf("Deleted %d\n",b);
        front=temp->next;
        free(temp);
        rear->next=front;
    }
}

//Function to show the peek of queue
//Time complexity => O(1)
void List::peek() {
```

```
    if(isempty()) {
        printf("Queue empty.");
    }

    else {
        int b=front->data;
        printf("Peak is %d\n",b);
    }
}

//Function to display the elements of the queue
//Time complexity => O(n)
void List::display() {
    struct Node *temp=front;
    if(isempty()) {
        printf("Queue empty.");
    }

    else {
        while(temp!=rear) {

            printf("%d,",temp->data);
            temp=temp->next;

        }
        printf("%d",temp->data);
    }
}
```

OUTPUT:

```
 CIRCULAR QUEUE

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 1
Enter the number 4

 Inserted successfully.
 CIRCULAR QUEUE

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 1
Enter the number 5

 Inserted successfully.
 CIRCULAR QUEUE

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 3
Peak is 4
```

```
 CIRCULAR QUEUE

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 2
Deleted 4

 CIRCULAR QUEUE

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 3
Peak is 5

 CIRCULAR QUEUE

 1. Enqueue
 2. Dequeue
 3. peek
 4. Exit

 Enter the choice 4
PROGRAM ENDED
```

Q)E)

AIM:

To Implement the round-robin scheduling algorithm using the circular queue ADT

ALGORITHM:

1.ENQUEUE:

Input: num (remaining CPU time for the task)

Output: 1 on successful insertion

1. Allocate memory for a new node.
2. Set the new node's data to num.
3. If the queue is empty (front is NULL):
   - Set newnode's next pointer to itself.
   - Set both front and rear pointers to this new node.
4. If the queue is not empty:
   - Set newnode's next pointer to front.
   - Update rear's next pointer to point to the new node.
   - Move the rear pointer to the new node.
5. Return 1 to indicate success.

2.DEQUEUE:

Output: 1 if a process is removed or moved to the end of the queue, 0 if the queue is empty.

1. Check if the queue is empty:
   - If true, return 0.
2. If the process at the front of the queue requires time less than or equal to the

timeslot:

- Remove the front node.

- If it was the only node, reset front and rear to NULL.

- Else, adjust front to the next node and rear's next to the new front.

- Free the removed node.

3. If the process requires more time than the timeslot:

- Calculate the remaining time after the current timeslot.

- Remove the front node.

- Enqueue the remaining time as a new task at the rear of the queue.

- Adjust the front to the next node and rear's next to the new front if not the only node.

4. Return 1 to indicate the task was processed.

TIME COMPLEXITY:

1.ENQUEUE:O(1)

2.DEQUEUE:O(1)

CODE:

Header:

```
//program to implement cpu timeslot
#include<stdio.h>
#include<stdlib.h>
#define timeslot 25
class queue
{

  struct node
  {

    int data;
    struct node *next;

  };
    struct node *front;
    struct node *rear;
```

```
  public:
  queue()
  {
   front=NULL;
   rear=NULL;

  }
  int enqueue(int num);
  int dequeue();

};

//Method to insert an element in the queue
int queue::enqueue(int num)
{
  struct node *newnode=(struct node *)malloc(sizeof(struct node));
  if(front==NULL)
  {
    newnode->data=num;
    newnode->next=newnode;
    front=newnode;
    rear=newnode;
    return 1;
  }
  else
  {
    newnode->data=num;
    newnode->next=front;
    rear->next=newnode;
    rear=newnode;
    return 1;
  }
}

// Method to delete an element from the queue
int queue:: dequeue()
{

  if(front==NULL)
  {
    return 0;
  }
  else
  {
    if(front->data-timeslot<=0)
    {
      if(front==rear)
      {
```

```
        front=NULL;
        rear=NULL;
        return 1;
      }
      else
      {
        struct node *temp=front;
        front=front->next;
        rear->next=front;
        free(temp);
        temp=NULL;
        return 1;
      }

    }
    else
    {
      int time;
      time=front->data-timeslot;
      if(front==rear)
      {
        front=NULL;
        rear=NULL;
      }
      else
      {
        struct node *temp=front;
        front=front->next;
        rear->next=front;
        free(temp);
        temp=NULL;
      }
      enqueue(time);
      return 1;
    }
  }
}
```

CPP:

```
#include<stdio.h>
#include<stdlib.h>
#include"QE.h"
int main()
{
    queue q1;
    int choice;
```

```
    char i;
    while(1)

    {
        printf("\n1) Insert");
        printf("\n2) Execute");
        printf("\n3) Exit");
        printf("\nEnter your choice:");
        scanf("%d",&choice);
        getchar();
        switch (choice)
        {
            case 1:
                int num1;
                printf("Enter the number to insert:");
                scanf("%d",&num1);
                if(q1.enqueue(num1))
                {
                printf("Element is inserted successfully");
                }
                else
                {
                printf("Operation failed!");
                }
                break;

            case 2:
                if(q1.dequeue())
                {
                printf("Element is removed successfully");
                }
                else
                {
                printf("The queue is empty!");
                }
                break;

            case 3:
                printf("PROGRAM ENDED");
                return 0;
        }
    }
}
```

OUTPUT:

```
1) Insert
2) Execute
3) Exit
Enter your choice:1
Enter the number to insert:50
Element is inserted successfully
1) Insert
2) Execute
3) Exit
Enter your choice:2
Element is removed successfully
1) Insert
2) Execute
3) Exit
Enter your choice:2
Element is removed successfully
1) Insert
2) Execute
3) Exit
Enter your choice:2
The queue is empty!
1) Insert
2) Execute
3) Exit
Enter your choice:3
PROGRAM ENDED
```

AIM:

To write a program to remove '+' and non '+' character in the left of '+' from a string

ALGORITHM:

CHECK:

Input: string (an array of characters), s1 (pointer to the first stack), s2 (pointer to the second stack)

Output: Stack - The stacks are modified directly

1. Loop through each character in the string:
   a. If the character is not '+':
      - Push the character onto stack s1.
   b. If the character is '+':
      - Check if the next node in s1 is not NULL.
      - If it's not NULL, pop the top character from s1.
2. After processing all characters in the string:
   - While s1 is not empty:
      - Pop a character from s1.
      - Push the popped character onto s2.

End

TIME COMPLEXITY:

Check:O(1)

CODE:

/*

Take a string from the user that consists of the '+' symbol. Process the string such that the final string does not include the '+' symbol and the immediate left non-'+' symbol. Select and choose the optimal ADT. Implement the program by including the appropriate header file.

```
*/
#include<stdio.h>
#include<stdlib.h>
#include"QF.h"
int main() {
  List l1;
  int num;
  printf("Enter length of input string:");
  scanf("%d",&num);
  char string[num];
  printf("Enter input string:");
  scanf("%s",string);

  for(int i=0;i<num;i++) {
    if(l1.isempty()) {
      l1.push(string[i]);
      continue;
    }

    if(string[i]=='+') {
```

```
            l1.popreturn();
        }

        else {
            l1.push(string[i]);
        }
    }
    int strlen=-1;
    while(l1.isempty()!=1) {
        strlen++;
        string[strlen]=l1.popreturn();
    }
    printf("\nThe string is \n");
    for(int i=strlen;i>=0;i--) {
        printf("%c",string[i]);
    }
    return 0;
}
```

OUTPUT:

```
Enter length of input string:12
Enter input string:45fgd+++ab+c

The string is
45ac
```

WEEK 8- TREE ADT-BINARY TREE

Date-20/03//24

AIM:

To Write a separate C++ menu-driven program to implement Tree ADT using a character binary tree

ALGORITHM:

1.INSERT:

Input: num,root

Output: 0 or 1

If root = NULL

    Root=newnode

    Return 1

Else

    Repeat untill node is inserted

    If choice= 1

        Go left

    If choice= 2

        Go right

2.DELETE:

Input: root

Output: Tree -

If root=NULL

Return 0

Else if 1 child:

Set temp=child

Free node

Return temp

Else if 2 children

Set leftmost= rightchild

Repeat until leftmost->left=NULL/

Leftmost=leftmost->left

Leftmost->left=leftchild

Return rightchild

Else

Call delete(root->left or root->right)

3.INORDER:

Input : root

Output : Displays all the elements

If root=NULL
  Return;
Else
  Call inorder(root->left)
  Display root->data
  Call inorder(root->right)

4.POSTORDER:

Input : root

Output : Displays all the elements

If root=NULL
  Return
Else
  Call postorder(root->left)
  Call postorder(root->right)

Display root->data

5.PREORDER:

Input : root

Output : displays all the elements

```
if root=NULL
  return
else
  Display root->data
  call preorder(root->left)
  call preorder(root->right)
```

6.SEARCH:

Input : num,root

Output : true or false

```
if root = NULL
  return false
else if root->data=num
  return true
else
  call search(num,root->left) or search(num,root->right)
```

TIME COMPLEXITY:

1.INSERT: O(n)

2.DELETE:O(n)

3.INORDER:O(n)

4.POSTORDER:O(n)

5.PREORDER:O(n)

6.SEARCH:O(n)

CODE:

```
//Program to inplement tree data structure using linked list adt
#include<stdio.h>
#include<stdlib.h>

class Tree
{
   private:

      struct node
      {
         int data;
         struct node *left;
         struct node *right;
      };

      struct node *root;

      int recins(struct node *temp,struct node *newnode)
      {
         int loc;
         printf("\n Enter left(0) or right(1) ");
         scanf("%d",&loc);

         if(loc==0)
         {
            if(temp->left==NULL)
            {
               temp->left=newnode;
               return 1;
            }
```

```
      temp=temp->left;
    }
    else if(loc==1)
    {
      if(temp->right==NULL)
      {
        temp->right=newnode;
        return 1;
      }
      temp=temp->right;
    }
    return recins(temp,newnode);
}

int recpre(struct node *temp)
{
    if(temp==NULL)
    {
      return 1;
    }
    printf("%d\n",temp->data);
    recpre(temp->left);
    recpre(temp->right);
    return 1;
}

int recpost(struct node *temp)
{
    if(temp==NULL)
    {
```

```
    return 1;
  }
  recpost(temp->left);
  recpost(temp->right);
  printf("%d\n",temp->data);
  return 1;
}

int recin(struct node *temp)
{
  if(temp==NULL)
  {
    return 1;
  }

  recin(temp->left);
  printf("%d\n",temp->data);
  recin(temp->right);
  return 1;
}

int recsearch(struct node *temp, int num)
{
  if(temp==NULL)
  {
    return 0;
  }
  if(temp->data==num)
  {
    return 1;
```

```
    }
    if(recsearch(temp->left,num))
    {
        return 1;
    }
    if(recsearch(temp->right,num))
    {
        return 1;
    }
    return 0;
}

int recrecdel(struct node *temp)
{
    if(temp==NULL)
    {
        return 1;
    }
    recrecdel(temp->left);
    recrecdel(temp->right);
    free(temp);
    return 1;
}

int recdel(struct node *temp,struct node *prev)
{
    int loc;
    printf("\n Enter the number to delete ");
    scanf("%d",&loc);
```

```
if(loc==0)
{
   if(temp->left==NULL)
   {
      return 0;
   }
   prev=temp;
   temp=temp->left;
}
else if(loc==1)
{
   if(temp->right==NULL)
   {
      return 0;
   }
   prev=temp;
   temp=temp->right;
}
else if(loc==2)
{
   if(prev->left==temp)
   {
      prev->left=NULL;
   }
   else if(prev->right==temp)
   {
      prev->right=NULL;
   }
   else if(prev==temp)
   {
```

```
                prev=NULL;
            }
            return(recrecdel(temp));
        }
        return(recdel(temp,prev));
    }

  public:
    Tree()
    {
        root=NULL;
    }


    int insert(int);
    int inorder();
    int preorder();
    int postorder();
    int search(int);
    int deletion();
};

int main()
{
  Tree t1;
  int choice,num;
  while(1)
  {
    printf("Enter your choice: \n1.Insert \n2.Preorder \n3.Inorder 
\n4.Postorder \n5.Search \n6.Delete \n7.Exit");
    scanf("%d",&choice);
    switch(choice)
```

```
{
   case(1):
      printf("\n Enter the number to insert:");
      scanf("%d",&num);
      t1.insert(num);
      break;
   case(2):
      t1.preorder();
      break;
   case(3):
      t1.inorder();
      break;
   case(4):
      t1.postorder();
      break;
   case(5):
      printf("\n Enter the number to search:");
      scanf("%d",&num);
      if(t1.search(num))
      {
         printf("The element is present in the tree");
      }
      else
      {
         printf("The element is not present in the tree");
      }
      break;
   case(6):
      if(t1.deletion())
      {
```

```
                printf("The element is deleted");
            }
            else
            {
                printf("End of tree");
            }
            break;
        case(7):
            printf("PROGRAM ENDED");
            exit(0);
            break;
        }
    }
    return 1;
}

//Method to insert value
int Tree::insert(int num)
{
    struct node *newnode = (struct node *)malloc(sizeof(struct node));
    newnode->data=num;
    newnode->left=NULL;
    newnode->right=NULL;

    if(root==NULL)
    {
        root=newnode;
        return 1;
    }
```

```
    return recins(root,newnode);
}

//Method for preorder
int Tree::preorder()
{
    return recpre(root);
}

//Method for inorder
int Tree::inorder()
{
    return recin(root);
}

//Method for postorder
int Tree::postorder()
{
    return recpost(root);
}

//Method for searching
int Tree::search(int num)
{
    return recsearch(root,num);
}

//Method for deletion
int Tree::deletion()
{
```

```
    return recdel(root,root);
}
```

OUTPUT:

```
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit1

 Enter the number to insert:1
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit1

 Enter the number to insert:2

 Enter left(0) or right(1) 1
```

```
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit1

 Enter the number to insert:3

 Enter left(0) or right(1) 0
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit2
1
3
2
```

```
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit3
3
1
2
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit4
3
2
1
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit5

 Enter the number to search:2
The element is present in the tree
```

```
Enter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit6

 Enter the number to delete 2
The element is deletedEnter your choice:
1.Insert
2.Preorder
3.Inorder
4.Postorder
5.Search
6.Delete
7.Exit7
PROGRAM ENDED
```

Q)B)

AIM:

To Add a "construct expression tree" method to the binary tree data structure

ALGORITHM:

1) **INSERT:**

Input : string,root

Output : Tree

```
For char in string
    If char Is a operator
        Newnode->left=pop
        Newnode->right=pop
    Else
        Assign char to a newnode
    Push newnode
```

2.INORDER:

Input : root

Output : Displays all the elements

```
If root=NULL
   Return;
Else
   Call inorder(root->left)
   Display root->data
   Call inorder(root->right)
```

3.POSTORDER:

Input : root

Output : Displays all the elements

```
If root=NULL
```

```
    Return
  Else
    Call postorder(root->left)
    Call postorder(root->right)
    Display root->data
```

4.PREORDER:

Input : root

Output : displays all the elements

```
  if root=NULL
    return
  else
    Display root->data
call preorder(root->left)
call preorder(root->right)
```

TIME COMPLEXITY:

1. **INSERT:O(n)**
2. **INORDER:O(n)**
3. **PREORDER:O(n)**
4. **POSTORDER:O(n)**

CODE:

HEADER:

```
#include <stdio.h>

#include <stdlib.h>

#include "q2stack.h"


int checkoperator(char ch) {

  if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {

    return 1;

  }

  return 0;

}
```

```
class BT {
   struct Node *root;

public:
   int constructedTree;
   BT() {
      root = NULL;
      constructedTree = 0;
   }

   void constructExpressionTree(char *);
   void preorder(struct Node *);
   void postorder(struct Node *);
   void inorder(struct Node *);
   struct Node *getRoot() {
      return root;
   };
};

void BT::constructExpressionTree(char *expression) {
   struct Node *newNode;
   Stack stack;
   for (int i = 0; expression[i] != '\0'; i++) {
      if (checkoperator(expression[i]) == 0) {
         newNode = (struct Node *)malloc(sizeof(struct Node));
         newNode->data = expression[i];
         newNode->left = NULL;
         newNode->right = NULL;
         stack.push(newNode);
```

```
    }
    else {
      newNode = (struct Node *)malloc(sizeof(struct Node));
      newNode->data = expression[i];
      newNode->right = stack.pop();
      newNode->left = stack.pop();
      stack.push(newNode);
    }
  }
  root = stack.pop();
  constructedTree = 1;
}

//Method to print using inorder traversal.
void BT::inorder(struct Node *root) {
  if (root == NULL) {
    return;
  }
  inorder(root->left);
  printf("%c ", root->data);
  inorder(root->right);
}

//Method to print using preorder traversal.
void BT::preorder(struct Node *root) {
  if (root == NULL) {
    return;
  }
  printf("%c ", root->data);
  preorder(root->left);
```

```
   preorder(root->right);
}

//Method to print using postorder traversal.
void BT::postorder(struct Node *root) {
   if (root == NULL) {
      return;
   }
   postorder(root->left);
   postorder(root->right);
   printf("%c ", root->data);
}
```

CPP FILE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "q2.h"

int main() {


   int choice;
   char expression[100];
   BT tree;
   while(1) {
      printf("\n1.Postfix Expression \n2.Construct Expression Tree \n3.Pre
Order \n4.Post Order \n5.In Order \n6.Exit");
      printf("\nEnter the choice");
      scanf("%d", &choice);

      switch(choice){
```

```
case 1:
  printf("Enter the postfix expression: ");
  scanf(" %s", &expression);
  break;
case 2:
  if(strlen(expression) == 0) {
    printf("Enter the postfix expression first\n");
    break;
  }
  else {
    tree.constructExpressionTree(expression);
    printf("Expression Tree Constructed\n");
    break;
  }
case 3:
  if(tree.constructedTree == 0) {
    printf("Construct the expression tree first\n");
    break;
  }
  else {
    printf("Pre Order: ");
    tree.preorder(tree.getRoot());
    break;
  }
case 4:
  if(tree.constructedTree == 0) {
    printf("Please construct the expression tree first\n");
    break;
  }
  else {
```

```
                printf("Post Order: ");
                tree.postorder(tree.getRoot());
                break;
            }
        case 5:
            if(tree.constructedTree == 0) {
                printf("Please construct the expression tree first\n");
                break;
            }
            else {
                printf("In Order: ");
                tree.inorder(tree.getRoot());
                break;
            }
        case 6:
            printf("PROGRAM ENDED\n");
            return 0;
        default:
            printf("Invalid choice\n");
        }
    }

    return 0;
}
```

OUTPUT:

```
1.Postfix Expression
2.Construct Expression Tree
3.Pre Order
4.Post Order
5.In Order
6.Exit
Enter the choice1
Enter the postfix expression: abc*+d/

1.Postfix Expression
2.Construct Expression Tree
3.Pre Order
4.Post Order
5.In Order
6.Exit
Enter the choice2
Expression Tree Constructed

1.Postfix Expression
2.Construct Expression Tree
3.Pre Order
4.Post Order
5.In Order
6.Exit
Enter the choice3
Pre Order: / + a * b c d
```

```
1.Postfix Expression
2.Construct Expression Tree
3.Pre Order
4.Post Order
5.In Order
6.Exit
Enter the choice4
Post Order: a b c * + d /
1.Postfix Expression
2.Construct Expression Tree
3.Pre Order
4.Post Order
5.In Order
6.Exit
Enter the choice5
In Order: a + b * c / d
1.Postfix Expression
2.Construct Expression Tree
3.Pre Order
4.Post Order
5.In Order
6.Exit
Enter the choice6
PROGRAM ENDED
```

Q)C)

AIM:

To identify the **optimal** ADT that can find a given number, its previous smaller element, and the next bigger element.

ALGORITHM:

1. Get the input from the user
2. Store it in the array
3. Traverse the array
4. Repeat until array reaches null character
5. If element < key and >lower

   Lower=element
6. If element >key and < upper

   Upper=element

TIME COMPLEXITY: **O(n)**

CODE:

HEADER:

```
/*
C. Given 'n' numbers, identify the optimal ADT that you can find a given number, its previous smaller element, and the next bigger element. Implement the program by including the appropriate header file.
*/
#include <stdio.h>
#include <stdlib.h>

struct Node {
   int data;
   struct Node *left;
```

```
    struct Node *right;
};

class BST {
    struct Node* root;
    struct Node* newnode(int data);
    struct Node* insertnode(struct Node* root, int data);
    struct Node* Prev(struct Node* root, int target);
    struct Node* Next(struct Node* root, int target);

public:
    BST() {
        root = NULL;
    }

    void insert(int data);
    void PrevandNext(int target);
};

struct Node* BST::newnode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

struct Node* BST::insertnode(struct Node* root, int data) {
    if (root == NULL) return newnode(data);
    if (data <= root->data) {
```

```
        root->left = insertnode(root->left, data);
    }

    else {
        root->right = insertnode(root->right, data);
    }

    return root;
}

void BST::insert(int data) {
    root = insertnode(root, data);
}

void BST::PrevandNext(int target) {
    struct Node* prev = Prev(root, target);
    struct Node* next = Next(root, target);

    printf("To find %d\n", target);
    printf("Previous smaller element ");
    if (prev != NULL) {
        printf("%d\n", prev->data);
    }
    else {
        printf("Not found \n");
    }
    printf("Next greater element ");
    if (next != NULL) {
        printf("%d\n", next->data);
    }
```

```
    else {
        printf("Not found \n");
    }
}

struct Node* BST::Prev(struct Node* root, int target) {
    struct Node* prev = NULL;
    while (root != NULL) {
        if (root->data >= target) {
            root = root->left;
        }
        else {
            prev = root;
            root = root->right;
        }
    }
    return prev;
}

struct Node* BST::Next(struct Node* root, int target) {
    struct Node* next = NULL;
    while (root != NULL) {
        if (root->data <= target) {
            root = root->right;
        }
        else {
            next = root;
            root = root->left;
        }
    }
```

```
    return next;
}
```

CPP FILE:

```
#include <stdio.h>
#include "q3.h"

int main() {
    BST tree;
    int n, target;
    printf("Enter the number of elements ");
    scanf("%d", &n);
    printf("Enter the elements ");
    for (int i = 0; i < n; i++) {
        int num;
        scanf("%d", &num);
        tree.insert(num);
    }
    printf("Enter the number ");
    scanf("%d", &target);
    tree.PrevandNext(target);

    return 0;
}
```

OUTPUT:

```
Enter the number of elements 6
Enter the elements 1
2
3
4
5
6
Enter the number 4
To find 4
Previous smaller element 3
Next greater element 5
```

WEEK 9- TREE ADT-BINARY SEARCH TREE Date:27/03/24

AIM:

To Write a separate C++ menu-driven program to implement Tree ADT using a character binary tree

ALGORITHM:

1.INSERT:

Input: num

Output: 0 or 1

If root = NULL
    Root=newnode
    Return 1
Else
    Set temp=root
    Repeat until node is inserted
        If num<key
            Go left and insert if temp->left is NULL else repeat
        Else
            Go right and insert if temp->right is NULL else repeat

2.INORDER:

Input : root

Output : Displays all the elements

If root=NULL
    Return;
Else
    Call inorder(root->left)
    Display root->data
    Call inorder(root->right)

3.POSTORDER:

Input : root

Output : Displays all the elements

```
If root=NULL
  Return
Else
  Call postorder(root->left)
  Call postorder(root->right)
  Display root->data
```

4.PREORDER:

Input : root

Output : displays all the elements

```
if root=NULL
  return
else
  Display root->data
  call preorder(root->left)
  call preorder(root->right)
```

6.SEARCH:

Input : num,root

Output : true or false

```
if root = NULL
  return false
else if root->data=num
  return true
else
  if key<root
    call search(num,root->left)
  else
    call search(num,root->right)
```

TIME COMPLEXITY:

1.INSERT:O(n)

2.PREORDER:O(n)

3.INORDER:O(n)

4.POSTORDER:O(n)

5.SEARCH:O(n)

CODE:

```
/*
A. Write a separate C++ menu-driven program to implement Tree ADT using a
binary search tree. Maintain proper boundary conditions and follow good
coding practices. The Tree ADT has the following operations,

1. Insert
2. Delete
3. Preorder
4. Inorder
5. Postorder
6. Search
7. Exit

What is the time complexity of each of the operations?
*/

#include <stdio.h>
#include <stdlib.h>

class BST{
  private:
    struct Node{
```

```
            int data;
            struct Node*Left;
            struct Node* Right;
        };
        struct Node* Root;
        struct Node* NewNode;
        struct Node* Temp;
        struct Node* TempP;
        struct Node* createnode();
        void RecPreOrder(struct Node *);
        void RecInOrder(struct Node *);
        void RecPostOrder(struct Node *);
        int RecSearch(int,struct Node*);
    public:
        BST(){
            Root = nullptr;
            NewNode = nullptr;
        }
        int Insert(int);
        int Delete(int);
        void PreOrder();
        void InOrder();
        void PostOrder();
        int Search(int);

};


int main(){
    BST b;
    int opt=1,data;
```

```
while (opt!=0){
    printf("\n1)Insert\n2)Delete\n3)Preorder\n4)Inorder\n5)Postorder\n6)Sear
ch\n7)Exit\n");
    scanf("%d",&opt);
    switch (opt) {
    case 1:
        printf("Enter data to insert : ");
        scanf("%d",&data);
        if(b.Insert(data)) {
            printf("Inserted %d successfully\n",data);
        }
        break;
    case 2:
        printf("Enter data to Delete : ");
        scanf("%d",&data);
        if(b.Delete(data)) {
            printf("Delete %d successfully\n",data);
        }
        break;
    case 3:
        b.PreOrder();
        break;
    case 4:
        b.InOrder();
        break;
    case 5:
        b.PostOrder();
        break;
    case 6:
        printf("Enter data to search : ");
        scanf("%d",&data);
```

```
        if(b.Search(data)==1){
           printf("Found element inside the tree");
        }else{
           printf("This element is not found in the tree");
        }
        break;
      case 7:
      printf("PROGRAM ENDED\n");
        return 0;
      default:
        return 0;
        break;
      }
   }
}

struct BST::Node *BST::createnode(){
   return (struct Node*)malloc(sizeof(struct Node));
}

int BST::Insert(int val){
   NewNode = createnode();
   NewNode->data = val;
   NewNode->Left = nullptr;
   NewNode->Right = nullptr;
   if(Root==nullptr){
      Root = NewNode;
      return 1;
   }else{
      Temp = Root;
```

```
    while(1==1){
      if(Temp->data > val){
        if(Temp->Left == nullptr){
          Temp->Left = NewNode;
          return 1;
        }
        Temp = Temp->Left;
      }else{
        if(Temp->Right == nullptr){
          Temp->Right = NewNode;
          return 1;
        }
        Temp = Temp->Right;
      }
    }
  }
  return 0;
}

// Method to print data of the tree recursively and INORDER
// Time Complexity => O(n)
void BST::RecInOrder(struct Node *Root){
  if(Root->Left != nullptr){
    RecInOrder(Root->Left);
  }
  printf(" %d ",Root->data);
  if(Root->Right != nullptr){
    RecInOrder(Root->Right);
  }
}
```

```
// Method to print data of tree INORDER
// Time Complexity => O(n)

void BST::InOrder(){
   if(Root != nullptr){
      RecInOrder(Root);
   }
   else{
      printf("Tree is Empty");
   }
}

// Method to print data of the tree Recursively and PREORDER
// Time Complexity => O(n)

void BST::RecPreOrder(struct Node *Root){
   printf(" %d ",Root->data);
   if(Root->Left != nullptr){
      RecPreOrder(Root->Left);
   }
   if(Root->Right != nullptr){
      RecPreOrder(Root->Right);
   }
}

// Method to print data of tree PREORDER
// Time Complexity => O(n)

void BST::PreOrder(){
```

```
    if(Root != nullptr){
        RecPreOrder(Root);
    }
    else{
        printf("Tree is Empty");
    }
}

// Method to print data of the tree Recursively and POSTORDER
// Time Complexity => O(n)

void BST::RecPostOrder(struct Node *Root){
    if(Root->Left != nullptr){
        RecPostOrder(Root->Left);
    }
    if(Root->Right != nullptr){
        RecPostOrder(Root->Right);
    }
    printf(" %d ",Root->data);
}

// Method to print data of tree POSTORDER
// Time Complexity => O(n)

void BST::PostOrder(){
    if(Root != nullptr){
        RecPostOrder(Root);
    }
    else{
        printf("Tree is Empty");
```

```
    }
}

// Method to Delete a value form the tree
// Time Complexity => O(logn)
int BST::Delete(int val){
    struct Node* parent = nullptr;
    struct Node* current = Root;
    while(current != nullptr && current->data != val){
        parent = current;
        if(val < current->data)
            current = current->Left;
        else
            current = current->Right;
    }
    if(current == nullptr)
        return 0;
    if(current->Left == nullptr && current->Right == nullptr){
        if(current != Root){
            if(parent->Left == current)
                parent->Left = nullptr;
            else
                parent->Right = nullptr;
        } else {
            Root = nullptr;
        }
        return 1;
        free(current);
    }
    else if(current->Left == nullptr || current->Right == nullptr){
```

```
        struct Node* child = (current->Left != nullptr) ? current->Left : current-
>Right;
        if(current != Root){
            if(current == parent->Left)
                parent->Left = child;
            else
                parent->Right = child;
        } else {
            Root = child;
        }
        return 1;
        free(current);
    }
    else {
        struct Node* successor = current->Right;
        while (successor->Left != nullptr)
            successor = successor->Left;
        int temp = successor->data;
        Delete(temp);
        current->data = temp;
        return 1;
    }
    return 1;
}

int BST::RecSearch(int val,struct Node* root){
    if(root==nullptr){
        return 0;
    }else{
        if(root->data == val){
            return 1;
```

```
    }else{
      if(root->data > val){
        if(RecSearch(val,root->Left)){
          return 1;
        }else{
          return 0;
        }
      }
      else{
        if(RecSearch(val,root->Right)){
          return 1;
        }else{
          return 0;
        }
      }
    }
  }
}

int BST::Search(int val){
  if(RecSearch(val,Root)){
    return 1;
  }else{
    return 0;
  }
}
```

OUTPUT:

```
1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
1
Enter data to insert : 1
Inserted 1 successfully

1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
1
Enter data to insert : 2
Inserted 2 successfully

1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
1
Enter data to insert : 3
Inserted 3 successfully
```

```
1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
1
Enter data to insert : 4
Inserted 4 successfully

1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
2
Enter data to Delete : 4
Delete 4 successfully

1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
3
 1  2  3
```

```
1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
4
 1  2  3
1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
5
 3  2  1
1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
6
Enter data to search : 3
Found element inside the tree
```

```
1)Insert
2)Delete
3)Preorder
4)Inorder
5)Postorder
6)Search
7)Exit
7
PROGRAM ENDED
```

Q)B)

AIM:

To write a program to find the count of ideal substrings In a string

ALGORITHM:

CHECK:

1)start

2) Initialize a count variable to 0.
3) Loop through the characters of the provided string in steps of 3.
4) For each step, push three characters onto the stack s1.
5) Pop the three characters from the stack into variables a, b, and c.
6) Check if all three characters are different.
7) If they are different, increment the count.
8) After the loop, return the count
9) Stop

TIME COMPLEXITY:

CHECK: O(n)

CODE:

HEADER FILE:

/*

B. A substring is a contiguous sequence of characters in a string. An ideal sub-string has a length of 3 with no repeating characters. Identify the optimal ADT and data structure to count the ideal number of substrings given a string of length 'n'. Multiple occurrences of a substring can be counted.

*/

```
#include <stdio.h>
#include <stdlib.h>

class List{
   private:
      struct Node{
         char data;
         struct Node *Next;
      };

      struct Node *Head;
      struct Node *Tail;
      struct Node *CurrentPoint;
      struct Node *NewNode;
   public:
      struct First3{
         char F,M,L;
      };
      struct First3 *val;
      List(){
         val = (struct First3*)malloc(sizeof(struct First3));
         Head = nullptr;
      }
      struct First3 *firstThree();
      struct Node *CreateNode();
      void DeleteNode(struct Node *);
      int IsEmpty();
      int Insert_Beg_Node(char);
      int Append_Node(char);
      int Insert_Pos_Node(char,int);
```

```
        int Delete_Beg_Node();
        int Pop_Node();
        int Delete_Pos_Node(int);
        void Display();
        char DisplayFront();
        void recursiveDisp(struct Node *);
        void Rev_Display();
        void recrev(struct Node *);
        void recrev1();
};

//Function to reate new node
struct List::Node * List::CreateNode(){
    return (struct Node *)malloc(sizeof(struct Node));
}

//Function to delete node
void List::DeleteNode(struct Node *del){
    free(del);
}

//Function to check if the list is empty
int List::IsEmpty(){
    if (Head==nullptr){
        return 1;
    }
    else{
        return 0;
    }
```

```
}

//Function to insert node in the beginning
int List::Insert_Beg_Node(char val){
   NewNode = CreateNode();
   NewNode->data = val;
   NewNode->Next = Head;

   if(Head==nullptr) {
      Tail=NewNode;
   }
   Head = NewNode;
   return 1;
}

//Function to insert node in the end
int List::Append_Node(char val){
   if(IsEmpty()==0){
      NewNode = CreateNode();
      NewNode->data = val;
      NewNode->Next = nullptr;
      Tail->Next = NewNode;
      Tail=NewNode;
      return 1;
   }
   else {
      Insert_Beg_Node(val);
      return 1;
   }
   return 0;
```

```
}

//Function to insert node in the position
int List::Insert_Pos_Node(char val ,int pos){
   if(pos == 0){
      Insert_Beg_Node(val);
      return 1;
   }
   if(IsEmpty()==0){
      pos--;

      CurrentPoint = Head;
      while(pos >0){
         CurrentPoint = CurrentPoint->Next;
         pos--;
      }
      printf("%d",CurrentPoint->data);
      NewNode = CreateNode();
      NewNode->data = val;
      NewNode->Next = CurrentPoint->Next;
      CurrentPoint->Next = NewNode;
      return 1;
   }return 0;
}

//Function to delete node in the beginning
int List::Delete_Beg_Node(){
   if(IsEmpty()==0){
      CurrentPoint = Head;
      Head = CurrentPoint->Next;
```

```
        DeleteNode(CurrentPoint);
        return 1;

    }else{
        return 0;
    }
}

//Function to delete node in the end
int List::Pop_Node(){
    if(IsEmpty()==0){
        CurrentPoint = Head;
        NewNode = Head;
        while(CurrentPoint->Next != nullptr){
            NewNode = CurrentPoint;
            CurrentPoint = CurrentPoint->Next;
        }
        NewNode->Next = nullptr;
        DeleteNode(CurrentPoint);
        return 1;
    }else{
        return 0;
    }
}

//Function to delete node in the position
int List::Delete_Pos_Node(int pos){
    if(pos == 0){
        Delete_Beg_Node();
        return 1;
```

```
    }
    if(IsEmpty()==0){
        CurrentPoint = Head;
        NewNode = Head;
        while((CurrentPoint->Next != nullptr) && (pos>0)){
            pos--;
            NewNode = CurrentPoint;
            CurrentPoint = CurrentPoint->Next;
        }
        if(pos>0){
            return 0;
        }else{
            NewNode->Next = CurrentPoint->Next;
            DeleteNode(CurrentPoint);
            return 1;
        }
    }
    return 0;
}

//Display the contents of the List
void List::Display(){
    if(IsEmpty()==0){
        CurrentPoint = Head;
        printf("[");
        while(CurrentPoint->Next != nullptr){
            printf("%c,",CurrentPoint->data);
            CurrentPoint = CurrentPoint->Next;
        }
        printf("%c]",CurrentPoint->data);
```

```
    }else{
        printf("[]");
    }
}

char List::DisplayFront(){
    if(Head==nullptr){
        return '~';
    }
    return Head->data;
}

//Function to display recursively
void List::recursiveDisp(struct Node *NextNode){
    if(NextNode->Next==nullptr){
        printf("[%c,",NextNode->data);
    }else{
        char val = NextNode->data;
        recursiveDisp(NextNode->Next);
        printf("%c,",val);
    }
}

//Function to display the reverse list
void List::Rev_Display(){
    if(IsEmpty()==0){
        recursiveDisp(Head->Next);
        printf("%c]",Head->data);
    }else{
        printf("[]");
```

```
   }
}

//Function to reverse using recursion
void List::recrev(struct Node *NextNode){
   if(NextNode->Next == nullptr){
      Head = NextNode;
   }else{
      recrev(NextNode->Next);
      NextNode->Next->Next = NextNode;
      NextNode->Next = nullptr;
   }
}

void List::recrev1(){
   if(IsEmpty()==0){
      recrev(Head);
   }
}

struct List::First3 *List::firstThree(){
   if(Head!=nullptr && Head->Next !=nullptr && Head->Next->Next != nullptr){
      val->F = Head->data;
      val->M = Head->Next->data;
      val->L = Head->Next->Next->data;
      return val;
   }
   return nullptr;
}
```

CPP FILE:

```
/*

B. A substring is a contiguous sequence of characters in a string. An ideal
sub-string has a length of 3 with no repeating characters. Identify the optimal
ADT and data structure to count the ideal number of substrings given a string
of length 'n'. Multiple occurrences of a substring can be counted.

*/

#include <stdio.h>

#include <stdlib.h>

#include "Q2.h"


int main(){

   List L;

   struct List::First3 *a;

   char in;

   int c=0;

   printf("Enter the substring \n");

   getchar();

   while((in = getchar()) != EOF && in!='\n'){

      L.Append_Node(in);

   }

   while((a = L.firstThree()) != nullptr){

      if(a->F != a->L && a->F != a->M && a->L!= a->M){

         c++;

      }

      L.Delete_Beg_Node();

   }

   printf("Total unique substrings = %d",c);


}
```

OUTPUT:

```
Enter the substring
aababcabc
Total unique substrings = 4
```

WEEK 10 -PRIORITY QUEUE ADT-HEAP DATE-03/04/24

AIM:

To Write a separate C++ menu-driven program to implement Priority Queue ADT using a max heap.

ALGORITHM:

1.INSERT:

Input : num,arr,cur

Output : 0 or 1

```
If cur=size-1
   Return 0
Else if cur=-1
   Arr[cur++]=num
   Return 1
Else
   Cur++
   Arr[cur]=num
   Call heapfiy
   Return 1
```

2.HEAPIFY:

Input : arr,cur

Output : elements are heapified

```
Set temp=cur
Repeat until temp > 0
   Parent =i-1/2
   If arr[i]>arr[parent]
     Swap arr[i] and arr[parent]
```

3.DISPLAY:

Input : arr,cur

Output : elements

Set temp=cur

Repeat until temp >=0

Display arr[temp]

4.DELETE:

Input : arr,cur

Ouput : the max element

If cur=-1
Return -1
Else
Temp=arr[cur]
Arr[0]=arr[cur]
Call heapify
Return temp

5.SORT:

Input : arr,cur,queue

Output : sorted elements

Repeat until cur>=0
Queue->push(call delete)

Repeat until queue is empty
Queue->pop

6.SEARCH:

Input : num,arr,cur

Output : 0 or 1

```
If cur=-1
   Return 0
Else
   Set temp=0
   Repeat until temp<=cur
      If arr[temp]=num
      Return 1
      Temp++
   Return 0
```

TIME COMPLEXITY:

1.INSERT:O(nlogn)

2.HEAPIFY: O(logn)

3.DISPLAY:O(n)

4.DELETE:O(n)

5.SORT:O(nlogn)

6.SEARCH:O(n)

CODE:

/*

A. Write a separate C++ menu-driven program to implement Priority Queue ADT using a max heap. Maintain proper boundary conditions and follow good coding practices. The Priority Queue ADT has the following operations,

1. Insert

2. Delete

3. Display

4. Search

5. Sort (Heap Sort)

6. Exit

What is the time complexity of each of the operations?

*/

```
#include <stdio.h>
using namespace std;
#include<stdlib.h>
#include<queue>
#define SIZE 50
class heap {
   int arr[SIZE];
   int cur;
   public:
      heap() {
         cur = -1;
      }
   int insert(int num);
   void heapify();
   void display();
   int del();
   void sort(queue <int> q1);
   void displayq(queue <int> q1);
   int search(int num);


};

int main() {
   queue <int> q1;
   heap h1;
```

```
int choice;
int element;
int pos;
while(1) {

    printf("\n PRIORITY QUEUE ADT \n");
    printf("\n 1. Insert \n 2. Delete \n 3. Display \n 4. Search \n 5. Sort \n 6.
Exit");
    scanf("%d",&choice);

    switch(choice) {
      case 1:
        printf("Enter the number ");
        scanf(" %d",&element);
        if(h1.insert(element)) {
          printf("\n Inserted successfully.\n");
        }
        else {
          printf("\n Insertion unsuccessful.\n");
        }
        break;

      case 2:
        if(h1.del()) {
        printf("Element deleted successfully");
        }

        else {
        printf("The heap is empty");
        }
        break;
```

```
        case 3:
          h1.display();
          break;

        case 4:
          int num;
          printf("Enter the number to search");
          scanf("%d",&num);
          if(h1.search(num)) {
            printf("the number is in %d",h1.search(num));
          }

          else {
            printf("The element is not found.");
          }
          break;

        case 5:
          printf("SORTED LIST");
          h1.sort(q1);
          break;

        case 6:
         printf("PROGRAM ENDED ");
          return 0;

    }
  }
```

```
}

//Method to push into heap.
//Time complexity => O(logn).
int heap:: insert(int num) {

  if (cur==SIZE-1) {
    return 0;
  }

  else if(cur==-1) {
    arr[0]=num;
    cur++;
    return 1;
}
  else {
    cur++;
    arr[cur]=num;
    heapify();
    return 1;
  }
}

//Method for heapify.
void heap:: heapify() {
     int i=cur;
     while(i>0) {
        int parent=(i-1)/2;
        if(arr[i]>arr[parent]) {
          int temp=arr[i];
```

```
        arr[i]=arr[parent];
        arr[parent]=temp;
      }
      i--;

    }
  }

//Method to display the elements.
//Time complexity => O(n).
void heap::display() {

  for(int i=0;i<=cur;i++) {
    printf("%d\t",arr[i]);
  }
}

//Method to delete.
//Time complexity => O(logn).
int heap:: del() {
  if(cur==-1) {
    return '\0';
  }

  else {
    int temp=arr[0];
    arr[0]=arr[cur];
    cur--;
    heapify();
    return temp;
```

```
 }
}

//Method to sort the heap.
//Time complexity => O(nlogn).
void heap::sort(queue<int>q1) {
 while(cur!=-1) {
  q1.push(del());
 }
  displayq(q1);
}

void heap:: displayq(queue<int> q1) {
  queue<int>q2 = q1;
  while (!q2.empty()) {
    printf("\n%d\n",q2.front());
    q2.pop();
  }

}

//Method to search.
//Time complexity => O(n).
int heap:: search(int num) {
 for(int i=0;i<=cur;i++) {
  if(num==arr[i]) {
   return i+1;
  }
 }
 return 0;
```

}

OUTPUT:

```
 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit1
Enter the number
9

 Inserted successfully.

 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit1
Enter the number 18

 Inserted successfully.

 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit1
Enter the number 27

 Inserted successfully.
```

```
 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit1
Enter the number 45

 Inserted successfully.

 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit2
Element deleted successfully
 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit3
27      18      9
```

```
 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit4
Enter the number to search27
the number is in 1
 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit5
SORTED LIST
27

18

9

 PRIORITY QUEUE ADT

 1. Insert
 2. Delete
 3. Display
 4. Search
 5. Sort
 6. Exit6
PROGRAM ENDED
```

WEEK 11- HASH ADT -HASH TABLE

Date-10/04/24

AIM:

To Write a separate C++ menu-driven program to implement Hash ADT with Separate Chaining.

ALGORITHM:

1.INSERT:

Input : num,hashtable

Output : 0 or 1

```
Index= num mod size
If hashtable[index]=NULL
   Hashtable[index]=num
   Return 1
Else
   Set temp=hashtable[index]
   Repeat until temp->next!=NULL
        Temp=temp->next
   Temp->next=num
   Return 1
```

2.SEARCH:

Input : num,hashtable
Output: 0 or 1

```
Index=num mod size
If hashtable[index]=NULL
   Return 0
Else
   Set temp=hashtable[index]
   Repeat until temp->next!=NULL
       If temp->data=num
          Return 1
       Temp=temp->next
   Return 0
```

3.DELETE:

Input : num,hashtable

Output : 0 or 1

```
Index= num mod size
If hashtable[index]=NULL
   return 0
else if hashtable[index]=num
    hashtable[index]=temp->next
Else
   Set temp=hashtable[index]
   Repeat until temp->!=NULL && temp->data!=num
       Temp2=temp
       Temp=temp->next
   If temp=NULL
       Return 0
   If temp->data=num
       Temp2->next=temp->next
       Return 1
```

TIME COMPLEXITY:

1.INSERT- O(n)

2.DELETE:O(n)

3.SEARCH:O(n)

CODE:

/*

A. Write a separate C++ menu-driven program to implement Hash ADT with Separate Chaining. Maintain proper boundary conditions and follow good coding practices. The Hash ADT has the following operations,

1. Insert

2. Delete

```
3. Search
4. Exit

What is the time complexity of each of the operations?
*/

#include <stdio.h>
#include <stdlib.h>
#define size 10

class hash {
   struct Node {
      int data;
      struct Node *next;
   };
   struct Node *arr[size];
   void hashdisp(struct Node* head,int i) {
      struct Node *temp = head;

      if(head == NULL)  {
         printf("List empty in %d index\n",i);
      }

      else  {
         printf("Index %d : ",i);
         while(temp != NULL) {
            printf("%d ",temp->data);
            temp=temp->next;
         }
         printf("\n");
```

```
        }
    }

    public:
        hash() {
            for(int i=0;i<size;i++) {
                arr[i]=NULL;
            }
        }

        void insert(int);
        void disp();
        int deletion(int);
        int search(int);
};

int main() {
    hash h;
    int choice;
    int num;
    while(1) {

        printf("\n1. Insert \n2. Delete \n3. Search \n4. Exit");
        scanf("%d",&choice);

        switch(choice) {
            case 1:
                printf("Enter the number ");
                scanf("%d",&num);
                h.insert(num);
```

```
    break;

case 2:
    printf("Enter the number ");
    scanf("%d",&num);
    if(h.deletion(num)) {
        printf("\n Deleted successfully.");
    }
    else {
        printf("\n Deletion unsuccessful.");
    }
    break;

case 3:
    printf("Enter the number ");
    scanf("%d",&num);
    if(h.search(num)) {
        printf("\n Element exists in hash.");
    }
    else {
        printf("\n Element does not exist in hash.");
    }
    break;

case 4:
    printf("PROGRAM ENDED ");
    return 0;
    break;

case 5:
```

```
            h.disp();
            break;
        }
    }
}

//Method to insert into hash.
//Time complexity => O(n).
void hash::insert(int num) {
    struct Node *newnode = (struct Node *)malloc(sizeof(struct Node));
    newnode->data=num;
    newnode->next=NULL;
    int target=num%size;
    if(arr[target]==NULL) {
        arr[target]=newnode;
        printf("Inserted.");
    }
    else {
        struct Node *temp = arr[target];
        int count = 0;
        struct Node *prev = NULL;
        while(temp != NULL) {
            if(temp->data == num) {
                printf("The number exists.");
                count ++;
                break;
            }
            prev = temp;
            temp = temp->next;
        }
```

```
        if(count !=1) {
            if(temp == NULL) {
                prev->next = newnode;
            }
            else {
                temp->next = newnode;
            }
        }

    }
}

//Method to delete the key in the hash.
//Time complexity => O(n).
int hash::deletion(int num) {
    int target = num % size;
    if(arr[target]==NULL) {
        return 0;
    }
    if(arr[target]->data==num) {
        struct Node *temp = arr[target];
        arr[target]=arr[target]->next;
        free(temp);
        return 1;
    }

    struct Node *temp = arr[target];
    struct Node *prev = arr[target];
    while(temp != NULL) {
        if(temp->data==num) {
```

```
            prev->next=temp->next;
            free(temp);
            return 1;
        }
        prev = temp;
        temp = temp -> next;
    }
    return 0;
}

//Method to search the key in the hash.
//Time complexity => O(n).
int hash::search(int num) {
    int target=num%size;
    if(arr[target]==NULL) {
        return 0;
    }
    struct Node *temp = arr[target];
    while(temp != NULL) {
        if(temp->data==num) {
            return 1;
        }
        temp = temp -> next;
    }
    return 0;
}

//Method to display
void hash::disp() {
    for(int i=0;i<size;i++) {
```

```
        hashdisp(arr[i],i);
    }
}
```

OUTPUT:

```
1. Insert
2. Delete
3. Search
4. Exit1
Enter the number 9
Inserted.
1. Insert
2. Delete
3. Search
4. Exit1
Enter the number 18
Inserted.
1. Insert
2. Delete
3. Search
4. Exit1
Enter the number 27
Inserted.
1. Insert
2. Delete
3. Search
4. Exit2
Enter the number 9

 Deleted successfully.
1. Insert
2. Delete
3. Search
4. Exit3
Enter the number 27

 Element exists in hash.
```

```
1. Insert
2. Delete
3. Search
4. Exit4
PROGRAM ENDED
```

WEEK 12-GRAPH ADT   Date:15/04/24

AIM:

To Write a separate C++ menu-driven program to implement Hash ADT with Linear Probing.

ALGORITHM:

1)INSERTION:

```
Input: num,hashtable
Output: 1 if element is added else 0
index = num % size

If hashtable[index] is -1
    hashtable[index] = num
    Return 1
Else
    Loop from index to size-1
        If hashtable[index] is -1
            hashtable[index] = num
            Return 1
        Increment index
    End Loop
End If
Return 0
```

2)DELETION:

```
Input: num,hashtable
Output: 1 if number is deleted else 0
index = num % size
```

```
Loop from index to size-1
    If hashtable[index] equals num
        hashtable[index] = -1
        Return 1
    Else If hashtable[index] is -1
        Return 0
    End If
    Increment index
End Loop
Return 0
```

3)DISPLAY:

```
Input: hashtable
Output: all the elements are displayed
Loop from i=0 to size-1
    If hashtable[i] is not -1
        Print hashtable[i]
    End If
End Loop
```

4)SEARCH:

```
Input: num,hashtable
Output: 1 if element is found else 0
index = num % size

Loop from index to size-1
    If hashtable[index] equals num
        Return 1
```

    Else If hashtable[index] is -1
        Return 0
    End If
    Increment index
End Loop
Return 0

TIME COMPLEXITY:

1. INSERT: O(n)
2. DELETE: O(n)
3. SEARCH: O(n)

CODE:

/*

A. Write a separate C++ menu-driven program to implement Hash ADT with Linear Probing. Maintain proper boundary conditions and follow good coding practices. The Hash ADT has the following operations,

1. Insert

2. Delete

3. Search

4. Display

5. Exit

What is the time complexity of each of the operations?

*/

```
#include<stdio.h>
#include<stdlib.h>
#define size 25
```

```
class hash {
   int hashtable[size]={0};
   public:
   hash() {
     for(int i=0;i<size;i++) {
      hashtable[i]=-1;
     }
   }
   int insert(int num);
   void display();
   int del(int num);
   int search(int num);
};
int main() {
   hash h1;
   int choice;
   while(1) {
      printf("\n1) Insert");
      printf("\n2) Delete");
      printf("\n3) Search");
      printf("\n4) Display");
      printf("\n5) Exit");
      printf("\n Enter your choice ");
      scanf("%d",&choice);

      switch (choice) {
         case 1:
            int num1;
            printf("Enter the number to insert");
            scanf("%d",&num1);
```

```
    if(h1.insert(num1)) {
    printf("%d is inserted successfully",num1);
    }
    else {
       printf("fail");
    }
    break;

case 2:
    int num2;
    printf("Enter the number to delete");
    scanf("%d",&num2);
    if(h1.del(num2)) {
       printf("%d is deleted successfully",num2);
    }
    else {
       printf("Element is not present");
    }
    break;

case 3:
    int num3;
    printf("Enter the element to search");
    scanf("%d",&num3);
    if(h1.search(num3)) {
       printf("Element is found");
    }
    else {
       printf("Element is not found");
    }
```

```
            break;

          case 4:
            h1.display();
            break;

          case 5:
            printf("PROGRAM ENDED");
            return 0;
        }

    }
}

//Method to insert the element in hashtable
//Time complexity => O(n)
int hash::insert(int num) {
  int index=num%size;

  if(hashtable[index]==-1) {
   hashtable[index]=num;
   return 1;
  }
  else {
    while(index!=size-1) {
      if(hashtable[index]==-1) {
         hashtable[index]=num;
         return 1;
      }
      index++;
```

```
    }
  }
  return 0;
}

//Method to display the elements in hashtable
//Time complexity => O(n)
void hash:: display() {
  for(int i=0;i<size;i++) {
   if(hashtable[i]!=-1) {
      printf("%d\t",hashtable[i]);
   }
  }
}

//Method to delete a element in hashtable
//Time complexity => O(n)
int hash :: del(int num) {
   int index=num%size;
   while(index<size-1) {
     if(hashtable[index]==num) {
       hashtable[index]=-1;
       return 1;
      }
     else if(hashtable[index]==-1){
       return 0;
      }
     index++;
   }
   return 0;
```

```
}

//Method to search an element in hashtable
//Time complexity => O(n)
int hash:: search(int num) {
   int index=num%size;

   while(index<size-1) {
      if(hashtable[index]==num) {
         return 1;
      }
      else if(hashtable[index]==-1) {
         return 0;
      }
      index++;
   }
   return 0;

}
```

OUTPUT:

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the number to insert1
1 is inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the number to insert2
2 is inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the number to insert3
3 is inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice2
Enter the number to delete2
2 is deleted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice3
Enter the element to search3
Element is found
```

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice4
1       3
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice5
PROGRAM ENDED
PS C:\Users\kesav\OneDrive
```

Q)B)

AIM:

To Write a separate C++ menu-driven program to implement Hash ADT with Quadratic Probing

ALGORITHM:

1. INSERT:

Input: num,hashtable

Output: 1 if element is added else 0

```
index = num % size

If hashtable[index] is -1
    hashtable[index] = num
    Return 1
Else
    i = 1
    While true
        index = index + i * i
        If index >= size
            Break
        If hashtable[index] is -1
            hashtable[index] = num
            Return 1
        Increment i
    End While
End If
Return 0
```

2. DELETE:

Input: num,hashtable

Output: 1 if number is deleted else 0

```
index = num % size
i = 0
```

```
While true
    If hashtable[index] equals num
        hashtable[index] = -1
        Return 1
    Else If hashtable[index] is -1
        Return 0
    End If
    index = index + i * i
    If index >= size
        Break
    Increment i
End While
Return 0
```

3. SEARCH:

```
Input: num,hashtable
Output: 1 if element is found else 0
index = num % size
i = 0

While true
    If hashtable[index] equals num
        Return 1
    Else If hashtable[index] is -1
        Return 0
    End If
    index = index + i * i
```

If index >= size

Break

Increment i

End While

Return 0

4.DISPLAY:

Input: hashtable

Output: elements are displayed

For i = 0 to size-1

If hashtable[i] is not -1

Print hashtable[i]

TIME COMPLEXITY:

1. **INSERT: O(n) in the worst case**
2. **DELETE: O(n) in the worst case**
3. **SEARCH: O(n) in the worst case**
4. **DISPLAY:O(n)**

CODE:

/*

B. Write a separate C++ menu-driven program to implement Hash ADT with Quadratic Probing. Maintain proper boundary conditions and follow good coding practices. The Hash ADT has the following operations,

1. Insert

2. Delete

3. Search

4. Display

5. Exit

What is the time complexity of each of the operations?

```
*/
#include<stdio.h>
#include<stdlib.h>
#define size 25
class hash {
   int hashtable[size]={0};
   public:
      hash() {
      for(int i=0;i<size;i++) {
         hashtable[i]=-1;
      }
      }
      int insert(int num);
      void display();
      int del(int num);
      int search(int num);
};
int main() {
   hash h1;
   int choice;
   while(1) {
      printf("\n1) Insert");
      printf("\n2) Delete");
      printf("\n3) Search");
      printf("\n4) Display");
      printf("\n5) Exit");
      printf("\n Enter your choice");
```

```
scanf("%d",&choice);

switch (choice) {
  case 1:
    int num1;
    printf("Enter the number to insert");
    scanf("%d",&num1);
    if(h1.insert(num1)) {
    printf("%d inserted successfully",num1);
    }
    else {
      printf("Insertion unsuccessful");
    }
    break;

  case 2:
    int num2;
    printf("Enter the number to delete");
    scanf("%d",&num2);
    if(h1.del(num2)) {
      printf("%d deleted successfully",num2);
    }
    else {
      printf("Element is not present");
    }
    break;

  case 3:
    int num3;
    printf("Enter the element to search");
```

```
            scanf("%d",&num3);
            if(h1.search(num3)) {
               printf("Element is found");
            }
            else {
               printf("Element is not found");
            }
            break;

         case 4:
            h1.display();
            break;

         case 5:
            printf("PROGRAM ENDED");
            return 0;
      }

   }
}

//Method to insert the element in hashtable.
int hash::insert(int num) {
  int index=num%size;

  if(hashtable[index]==-1) {
   hashtable[index]=num;
   return 1;
  }
  else {
```

```
      int i=1;
      while(index!=size-1) {
       if(hashtable[index]==-1) {
          hashtable[index]=num;
          return 1;
       }
       index=i*i +index;
       i++;
      }
   }
   return 0;
}

//Method to display the elements in hashtable.
void hash:: display() {
   for(int i=0;i<size;i++) {
    if(hashtable[i]!=-1) {
       printf("%d\t",hashtable[i]);
    }
   }
}

//Method to delete a element in hashtable.
int hash :: del(int num) {
    int index=num%size;
    int i=0;
    while(index<size-1) {
      if(hashtable[index]==num) {
         hashtable[index]=-1;
         return 1;
```

```
      }
      else if(hashtable[index]==-1) {
        return 0;
      }
      index=i*i +index;
      i++;
    }
    return 0;
}

//Method to search an element in hashtable.
int hash:: search(int num) {
    int index=num%size;
    int i=0;
    while(index<size-1) {
       if(hashtable[index]==num) {
          return 1;
       }
       else if(hashtable[index]==-1) {
          return 0;
       }
       index=i*i +index;
       i++;
    }
    return 0;

}
```

OUTPUT:

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the number to insert9
9 inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the number to insert18
18 inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the number to insert27
27 inserted successfully
```

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice2
Enter the number to delete9
9 deleted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice3
Enter the element to search27
Element is found
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice4
27      18
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice5
PROGRAM ENDED
```

Q)D)

AIM:

To Write a separate C++ menu-driven program to implement Graph ADT with an adjacency list.

ALGORITHM:

1.INSERT:

```
Input: u, v,list
Output: 1 if edge is added, 0 otherwise
Create newnode1 with data u and newnode2 with data v

For i from 0 to cur-1
    If adjacencylist[i] data equals u
        For k from 0 to cur-1
            If adjacencylist[k] data equals v
                Set temp to adjacencylist[i]
                While temp next is not NULL
                    Move temp to temp next
                End While
                Set temp next to newnode2
                Return 1
            End If
        End For
        Set adjacencylist[cur] to newnode2
        Increment cur
        Return 1
    End If
    If adjacencylist[i] data equals v
        Similar block as above for adding newnode1
```

```
    End If
End For

If u equals v
    Add self-loop only once to adjacencylist at cur
Else
    Add both newnode1 and newnode2 to adjacencylist at cur and cur+1
Increment cur by 2

Return 1
```

2.DELETE:

```
Input: v,list
Output: 1 if vertex is deleted, 0 otherwise
For i from 0 to cur-1
    If adjacencylist[i] data equals v
        Set adjacencylist[i] to NULL
        For k from i to cur-1
            Shift adjacencylist[k+1] to adjacencylist[k]
        End For
        Decrement cur
        Return 1
    End If
End For
Return 0
```

3.SEARCH:

```
Input: v
Output: 1 if vertex is found, 0 otherwise
```

```
For i from 0 to cur-1
    If adjacencylist[i] data equals v
        Return 1
    End If
End For
Return 0
```

4.DISPLAY:

```
Input:Adjacency matrix
Output: Print all vertices and their edges
For i from 0 to cur-1
    Print adjacencylist[i] data
End For
```

TIME COMPLEXITY:

1. INSERT: O(cur^2) because of the nested loop, but O(1) if adjacency list for u or v is already present

2. DELETE: O(cur) because it requires a single loop through the current list of vertices

3. SEARCH: O(cur) as it may need to look at each vertex in the worst case

4.DISPLAY:O(n)

CODE:

/*

C. Write a separate C++ menu-driven program to implement Graph ADT with an adjacency matrix. Maintain proper boundary conditions and follow good coding practices. The Graph ADT has the following operations,

1. Insert

2. Delete

3. Search

4. Display

5. Exit

What is the time complexity of each of the operations?

/*

C. Write a separate C++ menu-driven program to implement Graph ADT with an adjacency matrix. Maintain proper boundary conditions and follow good coding practices. The Graph ADT has the following operations,

1. Insert

2. Delete

3. Search

4. Display

5. Exit

What is the time complexity of each of the operations?

*/

```
#include<stdio.h>
#include<stdlib.h>
#define size 5
class hash {
  struct node {
    int data;
    struct node *next;
  };
  int cur;
  struct node *adjacencylist[size];
```

```
    struct node *head;
    public:
        hash() {
            for (int i=0;i<size;i++) {
                adjacencylist[i]=NULL;
            }
            cur=0;
        }
        int insert (int u,int v);
        void display();
        int search(int v);
        int del(int v);
};

int main() {
    hash h1;
    int choice;
    while(1) {
        printf("\n1) Insert");
        printf("\n2) Delete");
        printf("\n3) Search");
        printf("\n4) Display");
        printf("\n5) Exit");
        printf("\n Enter your choice");
        scanf("%d",&choice);

        switch (choice) {
            case 1:
                int v11,v12;
                printf("Enter the vertex 1:");
```

```
    scanf("%d",&v11);
    printf("Enter the vertex 2:");
    scanf("%d",&v12);
    if(h1.insert(v11,v12)) {
    printf("Element is inserted successfully");
    }
    else {
      printf("fail");
    }
    break;

  case 2:
    int v21;
    printf("Enter the vertex to be deleted");
    scanf("%d",&v21);
    if(h1.del(v21)) {
      printf("%d is deleted successfully",v21);
    }
    else {
      printf("Element is not found");
    }
    break;

  case 3:
    int v3;
    printf("Enter the vertex to search");
    scanf("%d",&v3);

    if(h1.search(v3)) {
      printf("Element is found");
```

```
            }
            else {
               printf("Element is not found");
            }
            break;

         case 4:
            h1.display();
            break;

         case 5:
            printf("PROGRAM ENDED");
            return 0;
      }

   }
}

//Method to insert the element in the graph
int hash:: insert(int v1,int v2) {
   struct node *newnode1=(struct node *)malloc(sizeof(struct node));
   struct node *newnode2=(struct node *)malloc(sizeof(struct node));

   newnode1->data=v1;
   newnode1->next=NULL;
   newnode2->data=v2;
   newnode2->next=NULL;

   for (int i=0;i<cur;i++) {
      if(adjacencylist[i]->data==v1) {
```

```
        for(int k=0;k<cur;k++) {
            if(adjacencylist[k]->data==v2) {
            struct node *temp=adjacencylist[i];
            while(temp->next!=NULL) {
            temp=temp->next;
            }
            temp->next=newnode2;
            return 1;
            }
        }
        adjacencylist[cur]=newnode2;
        cur++;
        return 1;
    }
    if(adjacencylist[i]->data==v2) {
        for(int k=0;k<cur;k++) {
            if(adjacencylist[k]->data==v1) {
            struct node *temp=adjacencylist[i];
            while(temp->next!=NULL) {
            temp=temp->next;
            }
            temp->next=newnode1;
            return 1;
            }
        }
        adjacencylist[cur]=newnode1;
        cur++;
        return 1;
    }
}
```

```
    if(v1==v2) {
        adjacencylist[cur]=newnode1;
        adjacencylist[cur]->next=newnode2;
        cur++;
        return 1;
    }
    adjacencylist[cur]=newnode1;
    adjacencylist[cur+1]=newnode2;
    cur+=2;
    return 1;
}

//Method to display in the graph
void hash:: display() {
   for(int i=0;i<cur;i++) {
    printf("%d\t",adjacencylist[i]->data);
   }
}

//Method to delete the element in the graph
int hash:: del(int v) {
    for (int i=0;i<cur;i++) {
        if(adjacencylist[i]->data==v) {
            adjacencylist[i]=NULL;
            for (int k=i;k<cur;k++) {
                adjacencylist[k]=adjacencylist[k+1];
            }
            cur--;
            return 1;
        }
```

```
    }
    return 0;
}

//Method to search in the graph
int hash::search(int v) {
    for (int i=0;i<cur;i++) {
        if(adjacencylist[i]->data==v) {
            return 1;
        }
    }
    return 0;
}
```

OUTPUT:

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the vertex 1:1
Enter the vertex 2:2
Element is inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice1
Enter the vertex 1:3
Enter the vertex 2:4
Element is inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice2
Enter the vertex to be deleted3
3 is deleted successfully
```

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice3
Enter the vertex to search4
Element is found
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice4
1       2       4
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice5
PROGRAM ENDED
```

Q)C)

AIM:

To Write a separate C++ menu-driven program to implement Graph ADT with an adjacency matrix.

ALGORITHM:

1.INSERT:

```
Input: v1, v2,matrix
Output: 1 if edge is added, 0 otherwise
If cur is greater than or equal to N
   Print "Vertex does not exist"
   Return 0
Else
   Set row to 0, col to 0
   For i from 1 to cur
      If adjacencymatrix[0][i] equals v1
         Set col to i
      If adjacencymatrix[i][0] equals v2
         Set row to i
   End For

   For i from 1 to cur
      If adjacencymatrix[0][i] equals v2
         Set col to i
      If adjacencymatrix[i][0] equals v1
         Set row to i
   End For

   If row is not 0 and col is not 0
```

```
        Set adjacencymatrix[row][col] and adjacencymatrix[col][row] to 1
        Return 1
    Else If row is not 0 and col is 0, or row is 0 and col is not 0
        Print "Please provide a valid vertex"
        Return 0
    Else
        Set adjacencymatrix[0][cur] to v1
        Set adjacencymatrix[cur][0] to v1
        Set adjacencymatrix[0][cur+1] to v2
        Set adjacencymatrix[cur+1][0] to v2
        Set adjacencymatrix[cur][cur+1] and adjacencymatrix[cur+1][cur] to 1
        Increment cur by 2
        Return 1
    End If
End If
```

2.DELETE:

```
Input: v,matrix
Output: 1 if vertex is deleted, 0 otherwise
If cur equals 1
    Print "The matrix is empty already"
    Return 0
End If

For i from 1 to cur
    If adjacencymatrix[0][i] equals v
        For j from i to cur
            Shift adjacencymatrix[0][j+1] left to adjacencymatrix[0][j]
        End For
    End If
```

```
    If adjacencymatrix[i][0] equals v
        For j from i to cur
            Shift adjacencymatrix[j+1][0] up to adjacencymatrix[j][0]
        End For
        Decrement cur
        Return 1
    End If
End For
```

3.SEARCH:

```
Input: v,matrix
Output: 1 if vertex is found, 0 otherwise
For i from 1 to cur
    If adjacencymatrix[0][i] equals v
        Return 1
    End If
End For
Return 0
```

4.DISPLAY:

```
Input: Adjacency matrix
Output: Adjacency matrix
For i from 0 to cur-1
    For j from 0 to cur-1
        Print adjacencymatrix[i][j]
    End For
    Print newline
End For
```

TIME COMPLEXITY:

1. INSERT: O(N)

2. DELETE: O(N)

3. SEARCH: O(N)

4.DISPLAY:O(N)

CODE:

```
/*

D. Write a separate C++ menu-driven program to implement Graph ADT with
an adjacency list. Maintain proper boundary conditions and follow good
coding practices. The Graph ADT has the following operations,

1. Insert
2. Delete
3. Search
4. Display
5. Exit

What is the time complexity of each of the operations?
*/

#include<stdio.h>
#include<stdlib.h>
#define N 6
class graph {

  int adjacencymatrix[N][N];
  int cur;
  public:
```

```
    graph() {
        for (int i=0;i<N;i++) {
            for (int j=0;j<N;j++) {
            adjacencymatrix[i][j]=0;
            }
        }
        cur=1;
    }
    int insert (int u,int v);
    void display();
    int del(int v);
    int search(int v);
};

int main() {
    graph g1;
    int choice;
    while(1) {
        printf("\n1) Insert");
        printf("\n2) Delete");
        printf("\n3) Search");
        printf("\n4) Display");
        printf("\n5) Exit");
        printf("\n Enter your choice ");
        scanf("%d",&choice);

        switch (choice) {
            case 1:
                int v1,v2;
                printf("Enter the vertex 1:");
```

```
  scanf("%d",&v1);
  printf("Enter the vertex 2:");
  scanf("%d",&v2);
  if(g1.insert(v1,v2)) {
  printf("vertex is inserted successfully");
  }
  break;

case 2:
  int v;
  printf("Enter the vertex to be deleted:");
  scanf("%d",&v);
  if(g1.del(v)) {
    printf("vertex is deleted successfully");
  }
  else {
    printf("vertex is not present");
  }
  break;

case 3:
  int v3;
  printf("Enter the vertex to search");
  scanf("%d",&v3);

  if(g1.search(v3)) {
    printf("vertex is found");
  }
  else {
    printf("vertex is not found");
```

```
            }
            break;

         case 4:
            g1.display();
            break;

         case 5:
            printf("PROGRAM ENDED");
            return 0;
      }

   }
}

//Method to insert in the graph
int graph:: insert(int v1,int v2) {
   if(cur>=N) {
      printf("Vertex does not exist");
      return 0;
   }
   else  {
      int row=0,col=0;
      for (int i=1;i<=cur;i++) {
         if(adjacencymatrix[0][i]==v1) {
            col=i;
         }
         if(adjacencymatrix[i][0]==v2) {
            row=i;
         }
```

```
        for (int i=1;i<=cur;i++) {
            if(adjacencymatrix[0][i]==v2) {
                col=i;
            }
            if(adjacencymatrix[i][0]==v1) {
                row=i;
            }
        }
        if(row!=0 && col!=0) {
            adjacencymatrix[row][col]=1;
            adjacencymatrix[col][row]=1;
            return 1;
        }
        else if(row!= 0 && col==0|| row==0 && col!=0) {
            printf("Please provide a valid vertex");
            return 0;
        }


        else {
            adjacencymatrix[0][cur]=v1;
            adjacencymatrix[0][cur+1]=v2;
            adjacencymatrix[cur][0]=v1;
            adjacencymatrix[cur+1][0]=v2;
            adjacencymatrix[cur][cur+1]=1;
            adjacencymatrix[cur+1][cur]=1;
            cur+=2;
            return 1;
        }
    }
}
```

```
}

//Method to display in the graph
void graph::display() {
   for (int i=0;i<cur;i++) {
      for (int j=0;j<cur;j++) {
         printf("%d\t",adjacencymatrix[i][j]);
      }
      printf("\n");
   }
}

//Method to delete in the graph
int graph:: del(int v) {
   if(cur==1) {
      printf("The matrix is empty already");
      return 0;
   }
   for (int i=1;i<=cur;i++) {
      if(adjacencymatrix[0][i]==v) {
         for (int j=i;j<=cur;j++) {
            adjacencymatrix[0][j]=adjacencymatrix[0][j+1];
         }
      }
      if(adjacencymatrix[i][0]==v) {
         for(int j=i;j<=cur;j++) {
            adjacencymatrix[j][0]=adjacencymatrix[j+1][0];
         }
         cur--;
      }
```

```
    }
    return 1;
}

//Method to search in the graph
int graph:: search(int v) {
    for (int i=1;i<=cur;i++) {
        if(adjacencymatrix[0][i]==v) {
            return 1;
        }
    }
    return 0;
}
```

OUTPUT:

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice 1
Enter the vertex 1:1
Enter the vertex 2:2
vertex is inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice 1
Enter the vertex 1:3
Enter the vertex 2:4
vertex is inserted successfully
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice 2
Enter the vertex to be deleted:3
vertex is deleted successfully
```

```
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice 3
Enter the vertex to search4
vertex is found
1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice 4
0       1       2       4
1       0       1       0
2       1       0       0
4       0       0       0

1) Insert
2) Delete
3) Search
4) Display
5) Exit
 Enter your choice 5
PROGRAM ENDED
```